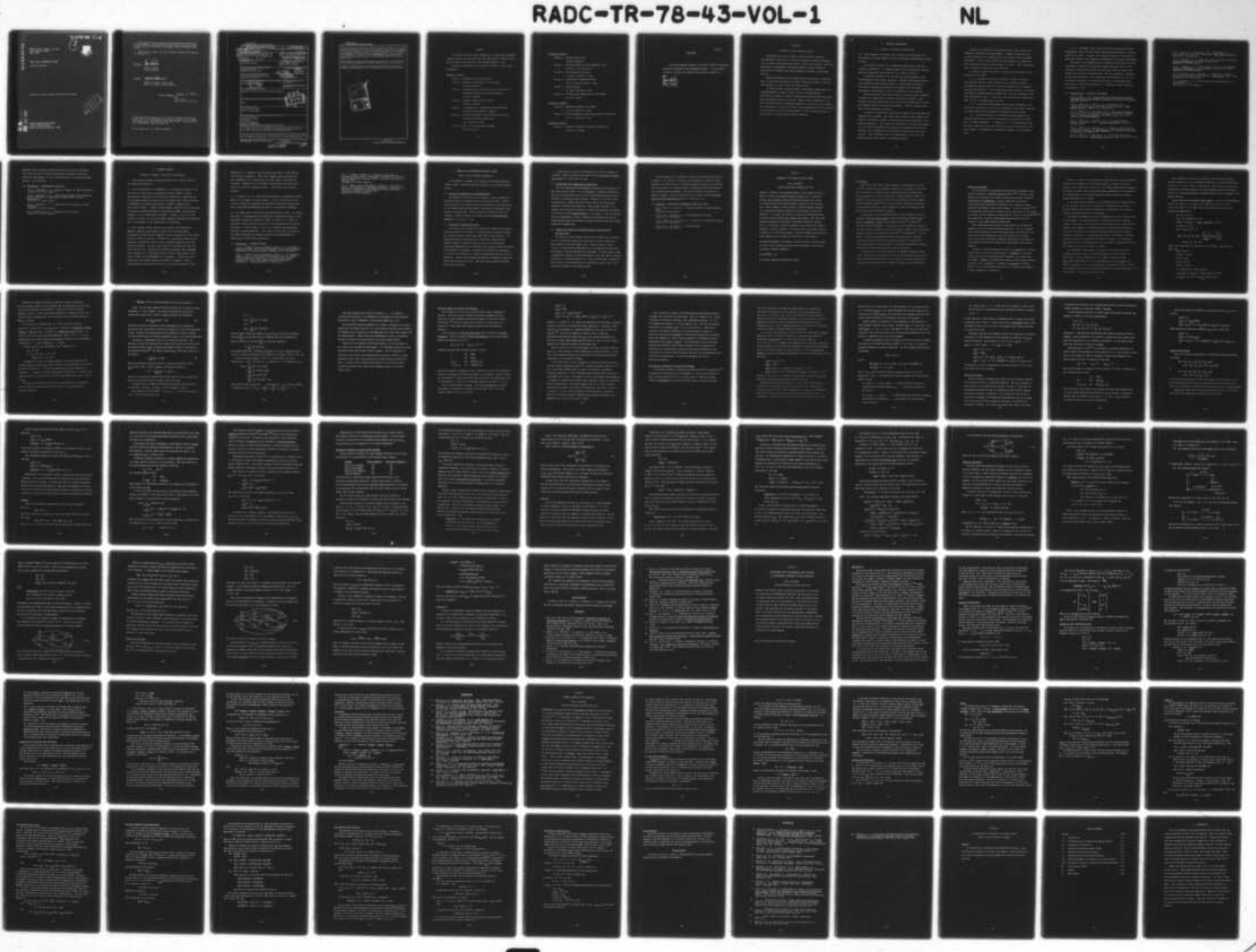
AD-A054 942 SYRACUSE UNIV N Y F/G 9/2
LARGE SCALE INFORMATION SYSTEMS. VOLUME I.(U)
MAR 78 F30602-74-C-0335
UNCLASSIFIED RADC-TR-78-43-VOL-1 NL

1 OF 2
AD
A054942



RADC-TR-78-43, Volume I (of four)
Final Technical Report
March 1978

LARGE SCALE INFORMATION SYSTEMS

Syracuse University

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-43, Volume I (of four) has been reviewed and is approved for publication.

APPROVED:

JAMES L. PREVITE
Project Engineer

APPROVED:

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
(18) 1. REPORT NUMBER: RADC-TR-78-43 Vol-1 (of four) ✓	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
(6) 4. TITLE (and Subtitle): LARGE SCALE INFORMATION SYSTEMS, Volume I.	(9) 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report 3 Jun 74 — 2 Feb 77
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Syracuse University	(15) 8. CONTRACT OR GRANT NUMBER(s): F30602-74-C-0335 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Syracuse University, Syracuse New York 13210	(16) 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810244 (17) P2
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCA), Griffiss AFB NY 13441	(11) 12. REPORT DATE: March 1978
	13. NUMBER OF PAGES: 146
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 144 p.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
James L. Previte (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel Processing
Programming Languages
Simulation
Data Base Management
Computer Architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes and references work conducted by Syracuse University in four broad areas: parallel processing, programming languages, modeling, and performance evaluation of generalized data management systems.

A number of applications were evaluated for processing by an associative processor architecture including air traffic control, carryless arithmetic and simulation of high-speed random logic. An extension of the typed lambda (Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339600

calculus has been developed which permits the binding and application of types. User-defined types and procedural data structures are shown to be complementary tools for data abstraction. Direct and continuation semantics of the domain of flow diagrams are formulated and the properties explored. The use of transition diagrams as a tool for structured programming has been investigated. A variety of concepts and notations have been devised to facilitate reasoning about arrays.

Work relation to various simulation tasks are reported on. A tutorial on the current statistical methods of analyzing simulation output data is provided.

A number of tasks relating to file systems are discussed and a framework is advanced for describing various file organizations and operations on files.

Preface

This report describes efforts completed in the Large Scale Information Systems project at Syracuse University under RADC contract F30602-74-C-0335. The work covers the period June 3, 1974 through February 2, 1977.

The report is produced in four volumes to facilitate single volume distribution.

Contents of Volume 1

Contents of Volume 2

Contents of Volume 3

Contents of Volume 4

3 Feb 78

EVALUATION

This effort advances concepts in the area of parallel processing, programming languages and data management systems. It also provides a tutorial and reports on simulation tasks which are specific to RADC.

JAMES L. PREVITE
Project Engineer

Section 1

Overview of the Contract Period

The objectives of the contract period were to explore developments in the use of large scale general purpose computers for Air Force data processing needs in four broad areas: parallel processing, programming languages, modeling and performance evaluation of generalized data management systems, and systems studies.

These developments were accomplished by principal investigators in each area working with other research personnel, faculty and students in each of the four broad areas.

In this section, each principal investigator summarises the accomplishments in each of their areas of investigation. References are made to interim reports published by RADC and other works produced during the period. Those unique references (i.e. not fully described in published reports), are part of this report. The references in each of the four areas summarized by the principal investigators give the section number of the referenced report in this report or references to other documents.

1. PARALLEL PROCESSORS

E. P. Stabler, Principal Investigator

1.1 This summary is divided into a series of descriptions of work done on various aspects of the Large Scale Information Systems Project.

Air Traffic Control is an important application area for parallel processing. A study was made of the processing requirements and parallel algorithms for air traffic control.

A diverse collection of low cost LSI microprocessors and microcomputers has become available. Modern engineering designs make extensive use of these items. A collection was made of descriptive engineering material on commercially available processors and systems to aid in evaluation of alternatives.

Sorting networks and various kinds of parallel data movement networks are important in parallel processing systems. Data movement and data positioning can cause bottle-necks which reduce the effectiveness of parallel processing. Several studies were made of data movement networks.

A system of mixed mode carryless arithmetic was developed and implemented on STARAN. The mixed mode system allows STARAN to do parallel arithmetic without incurring the delays associated with carry propagation. The study was continued to include development of 4 bit digit encoding and arithmetic processes which would be suitable for a STARAN-like configuration processing 4 bit data objects. The arithmetic unit for processing in this form was designed using high speed LSI processor chips.

A method of simulating high speed random logic networks on STARAN was developed and programmed. Random arrangements are the most difficult to simulate using the highly regular STARAN structure. In this case regularity was achieved by coding the gate functions and the interconnection as data. Very high speed logic simulation capability was obtained.

G. Foster joined the effort in June 1976 and worked on the following four tasks:

1.1.1 A study of an earlier effort to define a machine architecture for the direct execution of a subset of Space Programming Language Mark IV (SPL) was studied and an informal critique was prepared. The architecture description of the Space Programming Language Machine (SPLM) and its logic simulator (CLS) were described in APL. It was felt that non-trivial changes were made in SPL in implementing the SPLM.

1.1.2 A partial bibliography and selected readings in the literature pertaining to the direct execution high level languages has been prepared and informally submitted for evaluation.

1.1.3 Some assistance in following the developments of language constructs for multi-and parallel processing has been given to RADC personnel. In addition H. A. E. Spaanenburg, who has been associated with the contract until his termination this past summer, is independently pursuing research in this general area.

1.1.4 Readings in the areas of horizontal/vertical micro-programming trade offs have been undertaken preliminary to greater study of the QM-1. Further, some early findings in the areas of functional support for computing ensembles are now being sharpened.

The work on the STARAN system included the writing of an APL interpreter system which used the STARAN processor for all the arithmetic processing and data storage and using the PDP-11 for program storage and interpretation. The two processes, interpretation and arithmetic, were performed simultaneously in the two processors resulting in high utilization of the equipment and high speed. This work was done by J. Marzolf and was reported in reference 1.2.8.

1.2 Publications - PARALLEL PROCESSORS

1.2.1 Chen, I. N., "A Cellular Data Manipulating Array," Proceedings of 1975 Sagamore Conference on Parallel Processing, pp. 114-115.

1.2.2. Feng, T. Y., Chen, I. N., and Chen, Y. K., "Associative Processing of Network Flow Problems," RADC Report TR-76-93, March 1976, AD # A024 392.

1.2.3. Feng, T. Y. and Cheng, W. T., "Analysis and Design of a Cost Effective Parallel Processor for Weather Computations," Parallel Processing, Springer-Verlag, pp. 53-75. (Section 6 of this report.)

1.2.4. Feng, T. Y. and Hsu, C. P., "A Reconfigurable Parallel Arithmetic Unit," Parallel Processing, Srpinger-Verlag, 1975.

1.2.5. Feng, T. Y. and Lee, C. C., "Sorting Algorithms for Parallel Processing," Proceedings of 1975 Sagamore Conference on Parallel Processing, pp. 239-240.

1.2.6. Feng, T. Y. and Lee, C. C., "Parallel Multitonic Sorting Networks," RADC Report TR-76-93, March 1976, AD # 024392.

1.2.7. Feng, T. Y. and Yang, S. M., "An Approach of Developing Fast Transform Algorithms," RADC Report TR-76-92, March 1976 , AD # 9024 665.

1.2.8. Marzolf, J. G., "AAPL, An Array Processing Language," Parallel Processing, Springer-Verlag, 1975.
(Section 7 of this report.)

1.2.9. Stabler, E. P., "Mixed Mode Arithmetic for STARAN," Parallel Processing, Springer-Verlag, 1975, pp. 228-230.
(Section 14 of this report.)

1.2.10. Stabler, E. P. and Hsu, I., "Parallel Arithmetic Using Serial Arithmetic Processors," RADC Report, Sept. 1975.
(Section 15 of this report.)

1.2.11. Marzolf, J. G., "The Design and Implementation of APL-STARAN."
(Section 13 of this report.)

2. PROGRAMMING LANGUAGES

J. C. Reynolds, Principal Investigator

2.1 An extension of the typed lambda calculus has been developed which permits the binding and application of types. While retaining complete compile-time type checking, this language provides polymorphic functions and user-defined types. A semantic model has been formulated using Scott's lattice-theoretic approach. At present, we are trying to extend this language and its model to include the recursive definition of types. Reference 2.6.1.

2.2 User-defined types and procedural data structures have been shown to be complementary tools for data abstraction. Each approach has its own distinct advantages and limitations, and the two approaches cannot be unified, except by combining their limitations. Reference 2.6.2.

2.3 D. Scott has shown that a simple imperative language, in which primitive instructions are combined using sequencing and conditional operations, can be viewed as a lattice whose partial ordering denotes approximation, and that this lattice can be completed by permitting infinite programs, including the "unwindings" of iterative and recursive programs. We have formulated both the direct and continuation semant cs of this "domain of flow diagrams," and have explored their properties. The most important of these properties is that flow diagrams can be translated into a succession of increasingly "lower-level" infinitary languages, the last of which is canonical with respect

to continuation (but not direct) semantics. We have also shown that continuation semantics properly subsumes direct semantics. Reference 2.6.3.

2.4 The use of transition diagrams as a tool for structured programming has been investigated. The basic method is to represent all relevant states of knowledge about the state of the computation as nodes labelled with assertions, to represent each operation as an arc from its precondition to its post condition, and to represent each test as a complementary pair of arcs. The resulting transition diagram is easily translated into a program using goto operations. This methodology permits the systematic construction of useful programs which are difficult to impossible to construct with more conventional methods. It suggests that the hallmark of an intelligible transition diagram or flow chart is not its structural simplicity, but rather the simplicity of its assertions. No reference.

2.5 A variety of concepts and notations have been devised to facilitate reasoning about arrays. These include:

(a) Diagrammatic expressions denoting intervals of subscripts and partition relationships among such intervals.

(b) Pointwise extension of array values and relations.

(c) The generalization of the concept of ordering to arbitrary binary relations between array elements.

(d) A variety of equivalence relations between array values to be viewed as a monoid, or an associative monoid, under concatenation.

Examples such as binary search and various sorting algorithms show that the use of these notations can provide an order-of-magnitude improvement in the intelligibility of assertions about arrays. No references.

2.6 References - PROGRAMMING LANGUAGES

2.6.1. Reynolds, J. C., Towards a Theory of Type Structures."
(Section 4 of this report.)

2.6.2. Reynolds, J. C., "User-Defined Types and Procedural Data Structures as Complimentary Approaches to Data Abstraction."
(Section 3 of this report.)

2.6.3. Reynolds, J. C., "Semantics of the Domain of Flow Diagrams."
(Section 2 of this report.)

2.6.4 Reynolds, J. C., "Reasoning about Arrays."
(Section 8 of this report.)

3. SYSTEMS STUDIES

Robert G. Sargent, Principal Investigator

The system research performed under this contract will be discussed by projects.

3.1 The development of a SIMSCRIPT II.5 simulation model of the revised Air Force Military Personnel Center (AFMPC) Initial Operational Capability (IOC) Microfilm System was completed and the proposed system evaluated with the model during this contract. A SIMSCRIPT II.5 model operational on RADC's computer was delivered to RADC. A RADC Interim Report, RADC-TR-75-23, entitled "A Discrete Simulation Model of the Revised AFMPC IOC Microform System" (3.5.1.) describes the model. A paper (3.5.2.) on the project was given at the 1976 Summer Computer Simulation Conference and is contained in the Conference's Proceedings.

3.2 The largest effort during this contract was evaluating RADCOLS, Rome Air Development Center's On Line Simulator, developed for RADC by the Computer Science Corporation (CSC) under contract F30602-74-C-0281. RADCOLS is a simulation model of the Honeywell Information System (HIS) 600/6000 computers and GCOS Operating System. An error was found in the model and this was corrected by CSC. During this correction period, RADC upgraded their computer to a HIS 6000 system, a newer GCOS system, and a later version of the SIMSCRIPT II.5 compiler. Under this new combination the model would not execute or compile. After considerable evaluation and testing, an error was found in the

SIMSCRIPT II.5 compiler that occurred only when it was used on the HIS 6000 computers. CACI (the company that developed and owns SIMSCRIPT II.5) corrected the error under a new Release for RADC. RADCOLS will now compile and execute but then fails after it runs for a period of time. The reason is currently unknown.

3.3 A report (3.5.3.) "Statistical Analysis of Simulation Output Data" is in volume 3 of this report. It is a tutorial on the current statistical methods of analyzing simulation output data and it contains recommendations when to use specific methods.

3.4 Two additional activities were performed for RADC. An effort was begun to understand the transaction processing system of the Honeywell Information System in order to model it in order to determine better operating rules for this processor for specific workloads. A brief report (3.5.4.) on transaction processing systems is being submitted. The other activity was interacting with RADC personnel on modeling and testing methodology for system design and system evaluation.

3.5 References - SYSTEMS STUDIES

3.5.1. Trehan, Vijay and Sargent, Robert G., "A Discrete Simulation Model of the Revised AFMPC IOC Microform System," RADC-TR-75-23 Interim Report, February, 1975, AD # 007776

3.5.2. Trehan, Vijay and Sargent, Robert G., "A Simulation Model of the Air Force Military Personnel Center (AFMPC) Initial Operating Capability (IOC) Microform System," Proceedings of the 1976 Summer Computer Simulation Conference. (Not published in this report.)

3.5.3. Sargent, Robert G., "Statistical Analysis of Simulation Output Data," Department of Industrial Engineering and Operations Research Report, Syracuse University, January 1977.
(Section 16 of this report.)

3.5.4. Wood, Daniel and Sargent, Robert G., "An Introduction to Transaction Processing Systems," Department of Industrial Engineering and Operations Research Report, Syracuse University, January 1977.
(Section 5 of this report.)

4. MODELLING AND PERFORMANCE EVALUATION OF GDMS

Amrit L. Goel, Principal Investigator

The following is a summary of the research work pursued during the contract period. Details of this work are given in the various technical reports.

4.1 An Abstract Formulation of File Organizations

The purpose of this activity was to develop a rigorous framework for describing various file organizations and operations on files. Towards this objective, we first define various terms pertinent to the description of file organizations and then present a description of sequential, index sequential and hash files. Various operations on these files are also discussed. Details are given in the report entitled File Organization Concepts by Y. Liu and A. L. Goel.

4.2 Concurrency in Hashed File Access

In this part of the research we developed and studied a new technique, Concurrent Hashed Overflow (CHO), for improving the performance of hashed file systems in terms of both access counts and storage efficiency. The technique involves the exploitation of the ability of many existing computer systems to access more than one unit of data concurrently. Algorithms implementing CHO are presented and an analytical model describing its performance is developed. The problem of optimizing the performance of a CHO file is formulated as a dynamic programming problem and is solved numerically. Tables are presented giving dynamic programming optimum stage allocations and next stage state. From these tables optimum CHO designs may be easily obtained.

A paper based on this work was presented at the IFIP-74 Congress in Stockholm. Details are given in the report entitled Concurrency in Hashed File Access by L. Groner and A. L. Goel.

4.3 Cascade Hash File Organization for Disk Files

This file organization was developed to reduce the processing time for hash files stored on disks. It is a refinement of hashing and guarantees that all the probes after the first one are effectively sequential probes. Therefore the processing time is reduced. Moreover, for the same average processing time, a higher load factor may be used so that less storage overhead is required by the hash file. With a slight variation, it is possible to do the probes in parallel to reduce the processing time further. In the report entitled Cascade Hashing by Y. Liu and A. L. Goel, we describe this file organization in detail, formulate the optimization problem for optimal design, discuss the solution strategy and finally obtain the optimal designs for implementation.

4.4 Probabilistic Models for Processing Times on Index Sequential and Other Files

The objective of this research activity was to develop and illustrate the use of probabilistic models for times spent in various operations on index sequential and other simpler files. First we describe various criteria for evaluating the performance of a file organization and discuss the problem of tackling variability in performance. The basic models, using a probabilistic approach, are then developed for read, add, update and delete operations on an index sequential file. Based on these models, expressions are developed for processing times for sequential and direct files. Also discussed are models for some special cases.

The above models are not amenable to analysis by conventional statistical procedures. We, therefore, investigate various methods for numerical transformations of random variables, and choose the moments method for this problem. This method permits us to handle discrete, continuous and mixed distributions and is computationally manageable. The use of the models is described via numerical examples. This work is described in the report entitled Evaluation Models for Index Sequential Files by A. L. Goel and Y. Liu.

4.5 References - MODELLING AND PERFORMANCE EVALUATION OF GDMS

4.5.1. Goel, A. L. and Liu, Y., "Evaluation Models for Index Sequential Files."
(Section 9 of this report.)

4.5.2. Liu, Y. and Goel, A. L., "File Organization Concepts."
(Section 10 of this report.)

4.5.3. Groner, L. H. and Goel, A. L., "Concurrency in Hashed File Access."
(Section 11 of this report.)

4.5.4. Liu, Y. and Goel, A. L. "*Cascade Hashing.*"
(Section 12 of this report.)

Section 2

SEMANTICS OF THE DOMAIN OF FLOW DIAGRAMS

John C. Reynolds

Syracuse University, Syracuse, New York

ABSTRACT A domain of flow diagrams similar to that proposed by Scott, a domain of linear flow diagrams proposed by Goguen et al, a domain of decision table diagrams involving infinitary branching, and a domain of processes based on the ideas of Milner and Bekic are each provided with a direct semantics, closely related to partial-function semantics, and a continuation semantics similar to that developed by Morris and Wadsworth. It is shown that there are a variety of meaning-preserving continuous functions among these language-like domains, that every direct semantics possesses an "equivalent" continuation semantics, and that there is a particular continuation semantics which always gives distinct meanings to distinct processes. The proofs utilize the algebraic methods of Goguen et al, which are extended to continuous algebras with operations whose arguments can be indexed by infinite sets or even domains.

KEY WORDS AND PHRASES flow diagrams, lattices, domains, decision tables, processes, direct semantics, partial-function semantics, continuations, equivalence, algebraic semantics.

CR CATEGORIES 5.24

Work partly supported by NSF Grant GJ-41540.

Introduction

Scott[1] has shown that a simple language of flow diagrams, in which primitive instructions are combined by composition and conditional operators, can be embedded in a complete lattice containing infinite diagrams which include expansions of loops and recursions. Goguen, Thatcher, Wagner, and Wright[2] have shown that this lattice can be viewed as an initial algebra, so that algebraic methods[3] can be used to define and relate it semantics. They have also proposed a distinct algebra of "linear" flow diagrams with a more limited form of composition operator.

In this paper we will consider both kinds of flow diagram, as well as a domain of decision table diagrams, involving infinitary branching, and a domain of processes, suggested by the ideas of Milner[6] and Bekic.[10] For each of these "languages", we will define a direct semantics, similar to the partial-function semantics used in the theory of schemas, and a continuation semantics, similar to that developed by Morris[4] and Wadsworth.[5] We will exhibit a variety of meaning-preserving functions among these languages. We will also show that every direct semantics possesses an "equivalent" continuation semantics, and that there is a particular continuation semantics which always gives distinct meanings to distinct processes.

Our proofs will utilize and illustrate the algebraic methods of Goguen et al. To facilitate the treatment of decision table diagrams and processes, we will show that these methods can be extended to continuous algebras with operations whose arguments can be indexed by infinite sets or even domains.

Domains and Predomains

Our starting point is the so-called "lattice-theoretic approach" to the theory of computation, originally developed by Scott.[7,8] The heart of this approach is the assumption that domains of data can be partially ordered by a relationship $\sqsubseteq$ of approximation, in terms of which one can formulate a completeness property satisfied by the domains, and a continuity property satisfied by meaningful functions between domains. In Scott's own work,[1,7,8] the completeness property is the existence of least upper bounds for all subsets of a domain, so that a domain is a complete lattice. However, other authors[2,11-14] have worked with a variety of weaker completeness properties such as the existence of least upper bounds for directed sets.

After considerable experimentation, we have decided to work within the framework of one of the weaker completeness properties. The use of complete lattices would introduce so-called "overdefined" domain elements which do not possess any obvious computational reality and which preclude the simple formulation of several intuitively reasonable relationships, particularly the close connection between direct semantics and conventional partial-function semantics.

A subset of a partially ordered set is said to be directed iff it contains an upper bound for each of its finite subsets. A predomain is a partially ordered set in which every directed subset X possesses a least upper bound, written $\bigsqcup X$. A domain is a predomain which contains a least element, written $\perp$. A function f from a predomain S to a domain D is continuous iff $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all directed subsets X of S. A function from a domain to a domain is strict iff it preserves $\perp$.

There are a variety of ways in which we could alter these basic definitions with only minor changes in the ensuing development. We might use chains or countable chains instead of directed sets, or we might impose various topological restraints on domains, such as lattice continuity, algebracity, or the existence of countable bases. But the above definitions seem to provide the simplest adequate framework for our results.

In the terminology of Reference 2, our domains are strict Δ-complete posets, and our continuous functions are Δ-continuous. Perhaps the use of Scott's own term "domain" with a different definition is presumptuous, but we want to invoke the connotations of Scott's terminology.

The relatively unfamiliar concept of a predomain will play a central role in our development. An interesting portion of the lattice-theoretic approach extends to predomains, and they are useful intermediaries in the construction of domains. Most important, the concept includes both domains and ordinary sets, which we will consider to be predomains partially ordered by their identity relations. The result is a significant unification. For example, in defining semantics we will use an unspecified predomain S of states. In conventional applications S will be an ordinary set, but the validity of Propositions 7 and 8 will require the use of an S which is a domain.

For a predomain S and a domain D, we write $S \to D$ to denote the set of continuous functions from S to D, partially ordered by $f \sqsubseteq g$ iff $(\forall x \in S)$ $f(x) \sqsubseteq g(x)$. It can be shown that $S \to D$ is always a domain, in which $(\bigsqcup_{S\to D} F)(x) = \bigsqcup_D \{f(x) \mid f \in F\}$ for directed $F \subseteq S \to D$, and $\bot_{S\to D}(x) = \bot_D$. When S is a domain, $S \to D$ is the usual domain of continuous functions. At the other extreme, when S is a set (partially ordered by its identity relation), $S \to D$ is the domain of all functions from S to D.

For a predomain S, we write $S_\perp$ to denote the domain formed by adding a new least element to S, i.e., the disjoint union $\{\perp\} \cup S$, partially ordered by $x \sqsubseteq y$ iff $x = \perp$ or ($x \in S$ and $y \in S$ and $x \sqsubseteq_S y$). When S and S' are sets, there is an obvious isomorphism between $S \to S'_\perp$ and the domain of partial functions from S to S', partially ordered by the subset relation between the graphs of the partial functions.

We write Bool for the predomain {true, false}. Let S, S', S" be predomains, D, D', D" be domains, $e \in S \to S'_\perp$, $f \in S' \to S''_\perp$, $g \in S'' \to D$, $h \in D \to D'$, $i \in D' \to D''$, and $q \in S \to (S \to S''_\perp)$. Let ρ be a strict function in $D \to D'$. We define the expressions:

$$I_D \equiv \lambda x.\ x \in D \to D$$

$$h \cdot g \equiv \lambda s''.\ h(g(s'')) \in S'' \to D'$$

$$ext(g) \equiv \lambda x''.\ \underline{\text{if}}\ x'' = \perp\ \underline{\text{then}}\ \perp\ \underline{\text{else}}\ g(x'') \in S''_\perp \to D$$

$$J_S \equiv \lambda s.\ s \in S \to S_\perp$$

$$g^* f \equiv ext(g) \cdot f \in S' \to D$$

$$cond_D \equiv \lambda p.\ \lambda x_1.\ \lambda x_2.\ \underline{\text{if}}\ p = \left\{\begin{matrix}\perp\\ \underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\}\ \underline{\text{then}}\ \left\{\begin{matrix}\perp\\ x_1\\ x_2\end{matrix}\right\}$$

$$\in Bool_\perp \to (D \to (D \to D))$$

Each of these expressions is continuous in all variables. The reader may verify that

$$h \cdot I_D = I_{D'} \cdot h = h$$

$$(i \cdot h) \cdot g = i \cdot (h \cdot g)$$

$$ext(J_S) = I_{S_\perp}$$

$$e^* J_S = J_{S'}{}^* e = e$$

$$(g^* f)^* e = g^*(f^* e)$$

$$\lambda s.\ (g^*(q(s)))(s) = g^*(\lambda s.\ q(s)(s))$$

$$cond_{S'' \to D}(p, g_1, g_2)(s'') = cond_D(p, g_1(s''), g_2(s''))$$

$$\rho(cond_D(p, x_1, x_2)) = cond_{D'}(p, \rho(x_1), \rho(x_2))$$

As illustrated by the last two equations, we will often write $f(x_1, \ldots, x_n)$ for $f(x_1) \ldots (x_n)$.

Note that, under the isomorphism between $S \to S'_\perp$ and the domain of partial functions from a set S to a set S', the composition operator * mirrors the usual composition of partial functions.

For predomains S_1 and S_2, we write $S_1 \times S_2$ to denote the predomain $\{\langle x_1, x_2\rangle \mid x_1 \in S_1 \text{ and } x_2 \in S_2\}$, partially ordered by $\langle x_1, x_2\rangle \sqsubseteq \langle y_1, y_2\rangle$ iff $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$. When both S_1 and S_2 are domains, $S_1 \times S_2$ is a domain with the least element $\langle\perp, \perp\rangle$.

For predomains S_1 and S_2, we write $S_1 + S_2$ to denote the domain $\{\perp\} \cup \{\langle 1, x_1\rangle \mid x_1 \in S_1\} \cup \{\langle 2, x_2\rangle \mid x_2 \in S_2\}$, partially ordered by $x \sqsubseteq y$ iff $x = \perp$ or ($x = \langle i, x'\rangle$ and $y = \langle i, y'\rangle$ and $x' \sqsubseteq_{S_i} y'$). This is equivalent to defining $S_1 + S_2 = (S_1 \oplus S_2)_\perp$, where $\oplus$ denotes a conventional disjoint union *of sets* (with the obvious partial ordering).

We will need to generalize this kind of sum to an iterative construct. Suppose OP is a set and, for each $\sigma \in OP$, S_σ is a predomain. We write $\sum_{\sigma \in OP} S_\sigma$ to denote the domain $\{\perp\} \cup \{\langle \sigma, x'\rangle \mid \sigma \in OP \text{ and } x' \in S_\sigma\}$, partially ordered by $x \sqsubseteq y$ iff $x = \perp$ or ($x = \langle \sigma, x'\rangle$ and $y = \langle \sigma, y'\rangle$ and $x' \sqsubseteq_{S_\sigma} y'$).

When their operands are domains, + and $\sum$ denote the usual notion of a separated sum of domains.

If $f_\sigma \in S_\sigma \to D'$ for each $\sigma \in OP$, we write $\sum^*_{\sigma \in OP} f_\sigma$ to denote the function $g \in (\sum_{\sigma \in OP} S_\sigma) \to D'$ such that $g(\perp) = \perp$ and $g(\langle \sigma, x'\rangle) = f_\sigma(x')$.

Algebras and Homomorphisms

We will use a notion of algebra which is similar to the continuous algebras of Goguen et al.[2] However, we will generalize this notion to permit operations whose arguments can be indexed by arbitrary, perhaps infinite, predomains. On the other hand, we will not explore the many-sorted case treated in Reference 2, since the conventional one-sorted case is notationally simpler and adequate for our needs.

A signature Σ consists of a set OP of operators and a mapping rank which assigns a predomain to each operator in OP. For a predomain S, we write Σ_S to denote $\{\sigma \mid \sigma \in OP \text{ and } rank(\sigma) = S\}$. We will normally specify a signature by listing each nonempty Σ_S. A Σ-algebra ΣX consists of a domain X, called the carrier of ΣX, and an interpretation which assigns to each $\sigma \in \Sigma_S$ an operation $\Sigma X_\sigma \in (S \to X) \to X$.

In this formulation, conventional algebraic operations such as constant, unary, and binary operations are provided by the ranks S = {}, {1}, {1, 2}, ... Strictly speaking, we should write

$$\left.\begin{array}{l} \Sigma X_\sigma(<>) \\ \Sigma X_\sigma(<x_1>) \\ \Sigma X_\sigma(<x_1, x_2>) \end{array}\right\} \text{ instead of the conventional notation } \left\{\begin{array}{l} \Sigma X_\sigma \\ \Sigma X_\sigma(x_1) \\ \Sigma X_\sigma(x_1, x_2) \end{array}\right.$$

where $<x_1, \ldots, x_n>$ denotes the function from {1, ... , n} to X which maps i into x_i. However, we will frequently use the less cumbersome conventional notation. Since we regard $\Sigma X_\sigma(x_1, x_2)$ as an abbreviation for $\Sigma X_\sigma(x_1)(x_2)$, this is tantamount to identifying

$$(\{\} \to X) \to X \quad \text{with} \quad X$$

$$(\{1\} \to X) \to X \quad \text{with} \quad X \to X$$

$$(\{1,2\} \to X) \to X \quad \text{with} \quad X \to (X \to X).$$

Algebras will usually be named by giving their signature and carrier. One must remember however, that two algebras with the same signature and carrier can still have different interpretations. We will use Σ for an arbitrary signature, and Ω, Λ, Γ, and Δ for specific signatures. Similarly, we will use X (with occasional superscripts) for an arbitrary carrier and other symbols for specific carriers.

Let ΣX and $\Sigma X'$ be Σ-algebras and ρ be a strict continuous function from X to X'. If, for all $\sigma \in \Sigma_S$ and $\underset{\sim}{x} \in S \to X$, ρ satisfies the homomorphic equation $\rho(\Sigma X_\sigma(\underset{\sim}{x})) = \Sigma X'_\sigma(\rho \cdot \underset{\sim}{x})$, then ρ is said to be a homomorphism from ΣX to $\Sigma X'$. We write $\Sigma X \to \Sigma X'$ for the set of such homomorphisms. When $S = \{1, \ldots, n\}$, the identifications given above reduce the homomorphic equation to the conventional form $\rho(\Sigma X_\sigma(x_1, \ldots, x_n)) = \Sigma X'_\sigma(\rho(x_1), \ldots, \rho(x_n))$.

As usual, algebras and homomorphisms form a category, i.e.,

$I_X \in \Sigma X \to \Sigma X$

$\rho \in \Sigma X \to \Sigma X'$ and $\rho' \in \Sigma X' \to \Sigma X''$

implies $\rho' \cdot \rho \in \Sigma X \to \Sigma X''$.

An algebra ΣX is said to be initial (key) if, for each algebra $\Sigma X'$ with the same signature, there is exactly one (at most one) homomorphism from ΣX to $\Sigma X'$. As shown in Reference 2, all initial algebras with the same signature are isomorphic, so that we can speak of the initial algebra for Σ. We denote this algebra by $\Sigma In\Sigma$, its carrier by $In\Sigma$, and the unique homomorphism from it to $\Sigma X'$ by $\mu_{\Sigma X'}$.

Although these definitions of algebras and homomorphisms are unorthodox, one can still prove the following theorem, which is the sine qua non of algebraic semantics:

Theorem 1 There is an initial algebra $\Sigma In\Sigma$ for any signature Σ.

Proof: We will first construct $\Sigma In\Sigma$ and then show that it possess a unique homomorphism to any Σ-algebra. The carrier is obtained by using Scott's inverse limit construction to obtain a domain satisfying the isomorphism

$$In\Sigma \approx \sum_{\sigma \in OP} (rank(\sigma) \to In\Sigma) \qquad (1)$$

where OP and rank are the operator set and rank-mapping of the signature Σ. The inverse limit construction is described in detail in (9) and, more abstractly, in (15). Although these descriptions use the framework of complete lattices, the construction carries over without significant change to the present definitions.

In general, an isomorphism such as (1) can have many solutions. The particular solution $In\Sigma$ produced by the inverse limit construction (starting with the one-point domain as D_0) is uniquely characterized within an isomorphism by the following property:[9] The identity function $I_{In\Sigma}$ is the least solution of the equation

$$I = \sum^{*}_{\sigma \in OP} (\lambda \underset{\sim}{x}.\ \langle \sigma,\ I \cdot \underset{\sim}{x} \rangle) \qquad (2)$$

Here the parenthesized expression denotes a function from $rank(\sigma) \to In\Sigma$ to $\sum_{\sigma \in OP} (rank(\sigma) \to In\Sigma)$. Strictly speaking, (2) should be written as

$$I = \Phi^{-1} \cdot \left(\sum^{*}_{\sigma \in OP} (\lambda \underset{\sim}{x}.\ \langle \sigma,\ I \cdot \underset{\sim}{x} \rangle) \right) \cdot \Phi \ ,$$

where Φ is the isomorphic function from the left side of (1) to the right side, but we will adopt the practice of eliding Φ and its inverse.

To make $In\Sigma$ into a Σ-algebra, we provide the following interpretation of the operators: For all $\sigma \in \Sigma_S$ and $\underset{\sim}{x} \in S \to In\Sigma$, $\Sigma In\Sigma_\sigma(\underset{\sim}{x}) = \langle \sigma, \underset{\sim}{x} \rangle$.

Now, suppose ΣX is any Σ-algebra. Let $I_0, I_1, \ldots \in In\Sigma \to In\Sigma$ and $\mu_0, \mu_1, \ldots \in In\Sigma \to X$ be the functions such that

$$I_0 = \perp$$

$$I_{n+1} = \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \langle \sigma,\ I_n \cdot \underset{\sim}{x} \rangle)$$

$$\mu_0 = \perp$$

$$\mu_{n+1} = \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \Sigma X_\sigma(\mu_n \cdot \underset{\sim}{x})) \ .$$

By Scott's least fixed point theorem, the I_n's and μ_n's are directed sequences such that $\bigsqcup_{n=o}^{\infty} I_n$ is the least solution of equation (2), and is therefore the identity function for $In\Sigma$, while $\bigsqcup_{n=o}^{\infty} \mu_n$ is the least solution of

$$\mu = \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \Sigma X_\sigma(\mu \cdot \underset{\sim}{x})) \ .$$

This equation shows that μ satisfies $\mu(\Sigma In\Sigma_\sigma(\underset{\sim}{x})) = \mu(\langle \sigma, \underset{\sim}{x} \rangle) = \Sigma X_\sigma(\mu \cdot \underset{\sim}{x})$ for each operator σ, while the definition of $\sum^*$ insures that μ is strict. Thus $\mu \in \Sigma In\Sigma \to \Sigma X$.

On the other hand, suppose $\rho \in \Sigma In\Sigma \to \Sigma X$. We show that $\rho \cdot I_n = \mu_n$ by induction on n. For $n = 0$ we have $\rho \cdot \perp = \perp$, since $\rho(\perp) = \perp$. The induction step is

$$\begin{aligned}
\rho \cdot I_{n+1} &= \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \rho(\langle \sigma,\ I_n \cdot \underset{\sim}{x} \rangle)) \\
&= \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \rho(\Sigma In\Sigma_\sigma(I_n \cdot \underset{\sim}{x}))) \\
&= \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \Sigma X_\sigma(\rho \cdot I_n \cdot \underset{\sim}{x})) \\
&= \sum_{\sigma \in OP}^{*} (\lambda \underset{\sim}{x}.\ \Sigma X_\sigma(\mu_n \cdot \underset{\sim}{x})) = \mu_{n+1}
\end{aligned}$$

where the first line requires the strictness of ρ. But then the continuity of composition gives $\rho = \rho \cdot I_{In\Sigma} = \rho \cdot (\bigsqcup_{n=o}^{\infty} I_n) = \bigsqcup_{n=o}^{\infty} \rho \cdot I_n = \bigsqcup_{n=o}^{\infty} \mu_n = \mu$. □

When every operator has a rank of the form {1, ... , n}, Theorem 1 coincides with Corollary 4.10 of Reference 2, and despite its very different construction, $\Sigma In\Sigma$ is isomorphic to the initial algebra CT_Σ of (2).

The idea behind algebraic semantics is to regard a language as an initial algebra and its semantic function, i.e., the function which maps each element of the language into its meaning, as the homomorphism into some target algebra with the same signature. Indeed, since this homomorphism is unique, the semantic function is fixed by the specification of the target algebra itself.

However, in the framework of continuous algebras, an initial algebra is far richer than a conventional word algebra - our "languages" contain partially defined and (most mysteriously) infinite elements. But the imposition of strictness and continuity upon homomorphisms forces these elements to behave themselves, while their presence provides a profound capability explored by Scott: Concepts such as iteration and recursion can be viewed as purely syntactic mechanisms which permit finite language elements to abbreviate infinite ones.

The Direct Semantics of General Flow Diagrams

We now embark on a Cook's tour of several closely related languages and semantics. The initial view is informal; we provide a reasonable concrete syntax for the initial algebras, and describe semantics in the style of Scott and Strachey, [16] which, as illustrated in Section 3.2 of Reference 2, is equivalent to substituting target algebra operations into the homomorphic equations.

Let F be some set of primitive instructions and B be some set of Boolean expressions. Then the language of general flow diagrams is the initial algebra $\Omega In\Omega$ for the signature Ω such that

$$\Omega_{\{\}} = \{I\} \cup F, \qquad \Omega_{\{1,2\}} = \{;\} \cup B$$

Using the concrete syntax provided by Scott, we write:

I	for	$\Omega In\Omega_I$
f	for	$\Omega In\Omega_f$
$x_1; x_2$	for	$\Omega In\Omega_;(x_1, x_2)$
$b \rightarrow x_1, x_2$	for	$\Omega In\Omega_b(x_1, x_2)$

This initial algebra is isomorphic to CT_Σ, in Section 5.2, part II, of Reference 2. Except for the omission of overdefined elements, it is similar to Scott's lattice of flow diagrams. [1] (Albeit with other minor differences: Our formulation causes us to distinguish $\perp;\perp$ from $\perp$, and to disallow elements of the form $\perp \rightarrow x_1, x_2$.)

Let S be some predomain of states. Then the direct semantics of general flow diagrams is provided by the semantic function $\mu_{\Omega H} \in In\Omega \rightarrow H$, into the "semantic domain" $H = S \rightarrow S_\perp$, such that:

$$\mu_{\Omega H}(I) = J_S$$

$$\mu_{\Omega H}(f) = \mathcal{F}(f)$$

$$\mu_{\Omega H}(x_1; x_2) = \mu_{\Omega H}(x_2) * \mu_{\Omega H}(x_1)$$

$$\mu_{\Omega H}(b \rightarrow x_1, x_2) = \lambda s.\ \mathrm{cond}_{S_\perp}\ (\mathcal{B}(b, s), \mu_{\Omega H}(x_1, s), \mu_{\Omega H}(x_2, s))$$

Here $\mathcal{F} \in F \rightarrow H$ and $\mathcal{B} \in B \rightarrow (S \rightarrow \mathrm{Bool}_\perp)$ are unspecified functions which provide the meaning of primitive instructions and Boolean expressions. Informally, these equations can be regarded as a language definition in the style of Scott and Strachey.[16] But from the algebraic viewpoint, they are simply the homomorphic equations which assert that $\mu_{\Omega H}$ is the unique homomorphism from $\Omega \mathrm{In} \Omega$ into a certain target algebra ΩH with carrier H. The structure of target algebras such as ΩH will be given more abstractly later.

When S is a set, $H = S \rightarrow S_\perp$ is isomorphic to the set of partial functions from S to S, and * mirrors the usual composition of partial functions. In this case, direct semantics reduces to the usual kind of partial-function semantics encountered in the treatment of schemas.

On the other hand, our direct semantics is intentionally more restrictive than the semantics suggested by Scott,[1] in which the semantic domain is $S \rightarrow S$ for an unspecified complete lattice S, and * is replaced by conventional functional composition. Scott's semantics becomes unnatural when one does not require $\mathcal{F}(f)$ to be strict. For example, let δ be a flow diagram whose meaning is $\perp$, presumably a diagram whose execution never terminates. Then δ; f could have a different meaning than $\perp$, which would suggest that the statement following a nonterminating statement could affect the computation. The obvious solution is to replace $S \rightarrow S$ and $S \rightarrow \mathrm{Bool}_\perp$ by domains of strict functions. Our direct semantics is basically similar to imposing this strictness requirement, but it emphasizes that $\perp$ in $S_\perp$ is not really a "state".

Direct semantics is capable of describing schema-like languages involving assignment and side-effect free expressions. But it is inadequate for a variety of primitive instructions occurring in real programming languages. Let δ be a flow diagram whose meaning is $\perp$ (in the domain $S \to S_\perp$). Then for any primitive instruction f, the meaning $\mu_{\Omega H}(f; \delta)$ of f; δ will also be $\perp$. (Note that in contrast to the previous paragraph, we are now examining the effect of a primitive instruction which precedes a non-terminating diagram and can sensibly affect the computation.) This is clearly inadequate to accomodate primitive instructions such as stop or print(n). The introduction of such instructions precludes the assumption, made in direct semantics, that the meaning of an instruction is a function which accepts the state existing immediately prior to execution of the instruction and produces the state existing immediately after execution of the instruction. To avoid this assumption, we turn to continuation semantics.

The Continuation Semantics of General Flow Diagrams

In continuation semantics, originally developed by Morris[4] and Wadsworth,[5] the meaning of an instruction (or flow diagram) is a function which accepts the state existing immediately prior to execution, plus an additional argument called the continuation, and produces the final output of the entire program.

The continuation which is provided as a second argument is a function from the state existing after instruction execution to the final program output, which gives the semantics of the "rest of the computation" to be performed if the current instruction "terminates normally." Thus an instruction with normal behavior will produce its output by applying the continuation to the state following execution. But an "abnormal" instruction can produce the final output in some other manner - possibly ignoring the continuation.

Let S again be some predomain of states, and let O be some domain of outputs, whose least element $\perp$ denotes the "output" of a non-terminating computation. The domain of continuations is $C = S \to O$, and the semantic domain is $W = C \to (S \to O) = C \to C$. (Somewhat counterintuitively, we have made continuations the first argument and states the second argument of the meanings of flow diagrams; this will eventually simplify our semantic equations.) Then the continuation semantics of general flow diagrams is provided by the semantic function $\mu_{\Omega W} \in In\Omega \to W$ such that:

$$\mu_{\Omega W}(I) = \lambda c.\ \lambda s.\ c(s)$$

$$\mu_{\Omega W}(f) = \mathcal{G}(f)$$

$$\mu_{\Omega W}(x_1;\ x_2) = \lambda c.\ \lambda s.\ \mu_{\Omega W}(x_1,\ \lambda s'.\ \mu_{\Omega W}(x_2,\ c,\ s'),\ s)$$

$$\mu_{\Omega W}(b \to x_1, x_2) = \lambda c.\ \lambda s.\ cond_0(\mathcal{B}(b,s),\ \mu_{\Omega W}(x_1,\ c,\ s),\ \mu_{\Omega W}(x_2,\ c,\ s))$$

where $\mathcal{G} \in F \to W$ and $\mathcal{B} \in B \to (S \to Bool_\perp)$ are unspecified functions providing the semantics of primitive instructions and Boolean expressions.

The essence of continuation semantics is revealed by the third equation. Intuitively, to execute $x_1;\ x_2$ with an initial state s and a continuation c, we execute x_1 with the state s and a continuation $\lambda s'.\ \mu_{\Omega W}(x_2,\ c,\ s')$ which picks up the state s' after execution of x_1 and then executes x_2 with s' and the continuation c, which in turn picks up the state after execution of x_2 and then

executes the rest of the program. But more generally, the equation shows that it is the meaning $\mu_{\Omega W}(x_1)$ of x_1 which determines how the final output will be affected by the meaning $\mu_{\Omega W}(x_2)$ of x_2, which in turn determines how the final output will be affected by the "meaning of the rest of the program" c.

Further insight is provided by a brief digression on the semantics of some "abnormal" primitive instructions. To handle stop ε F, we can take $O = S_\perp$ and $\mathcal{G}$ (stop) = $\lambda c.\ \lambda s.\ s = \lambda c.\ J_S$. This makes it clear that the final output caused by a stop instruction will be the state existing immediately before its execution, regardless of the rest of the program.

To handle intermediate output of integers, let Int be the set of integers, N be some set of integer expressions, and $\mathcal{N} \varepsilon\ N \to (S \to Int_\perp)$ be a function giving the semantics of integer expressions. Let O be a domain satisfying the isomorphism

$$O \approx S + Int \times O$$

and let

$$\mathcal{G}\ (\text{print n}) = \lambda c.\ \lambda s.\ ext(\lambda i\ \varepsilon\ Int.\ \rho_2(i, c(s)))(\mathcal{N}(n, s))$$

$$\mathcal{G}\ (\text{stop}) = \lambda c.\ \lambda s.\ \rho_1(s)$$

where ρ_1 and ρ_2 are the obvious injection functions from S and Int × O, respectively, into O. The elements of O can be classified into three distinct kinds of output:

(1) $\rho_2(i_1, \ldots, \rho_2(i_k, \rho_1(s)) \ldots)$, which would be the output of a program which prints the integers $i_1, \ldots, i_k$ and then terminates in the state s.

(2) $\rho_2(i_1, \ldots, \rho_2(i_k, \perp) \ldots)$, which would be the output of a program which prints the integers $i_1, \ldots, i_k$ and then runs forever without further printing.

(3) $p_2(i_1, p_2(i_2, \ldots))$ - a limit point in the domain O - which would be the output of a program which prints the endless sequence of integers $i_1, i_2, \ldots$.

In this situation, "final" output is a misnomer, since a program can continue to generate output forever. A better term would be irreversible output, since the semantics insures that the rest of the program cannot rescind the effect of a print instruction.

Returning to general flow diagrams, where the interpretation of primitive instructions is left unspecified, we can simplify our equations for $\mu_{\Omega W}$ by using eta-reduction. Specifically, $\lambda s.\ c(s) = c$, $\lambda s'.\ \mu_{\Omega W}(x_2, c, s') = \mu_{\Omega W}(x_2, c)$, and $\lambda s.\ \mu_{\Omega W}(\ \ldots\ , s) = \mu_{\Omega W}(\ \ldots\ )$. Thus

$$\mu_{\Omega W}(I) = \lambda c.\ c = I_C$$

$$\mu_{\Omega W}(f) = \mathcal{U}(f)$$

$$\mu_{\Omega W}(x_1; x_2) = \lambda c.\ \mu_{\Omega W}(x_1, \mu_{\Omega W}(x_2, c)) = \mu_{\Omega W}(x_1) \cdot \mu_{\Omega W}(x_2)$$

$$\mu_{\Omega W}(b \rightarrow x_1, x_2) = \lambda c.\ \lambda s.\ \mathrm{cond}_0(\mathcal{B}(b,s), \mu_{\Omega W}(x_1,c,s), \mu_{\Omega W}(x_2,c,s))$$

Intriguingly, the order of composition in the third equation is the reverse of the order for direct semantics.

Linear Flow Diagrams

There are many flow diagrams which possess the same meaning, regardless of the choice of direct or continuation semantics. The other languages we shall consider can be thought of as successive attempts to strip away this redundancy and approach the goal of canonicality, where distinct diagrams have distinct meanings. Eventually, it will become clear that we have a succession of languages with meaning-preserving mappings from each language to the next, where the final language (of processes) is canonical for continuation semantics. For the present however, this image is only meant

to motivate our definitions, and no attempt will be made to prove relationships between languages or equivalences within a language.

In any reasonable semantics, one would expect the meaning of general flow diagrams to satisfy the following equivalences:

$$I;\ x_1 = x_1$$
$$(x_1;\ x_2);\ x_3 = x_1;\ (x_2;\ x_3)$$
$$(b \to x_1,\ x_2);\ x_3 = b \to (x_1;\ x_3),\ (x_2;\ x_3)$$

Intuitively - neglecting any complications which might be caused by infinite diagrams - these equivalences can be used to transform any flow diagram until every left operand of ";" is a primitive instruction. A flow diagram which meets this restriction is said to be <u>linear</u>.

Following Goguen et al,[2] we can formulate linear flow diagrams as an initial algebra by regarding f; x as the application of a unary operator, named by f, to the operand x. Thus the language of linear flow diagrams is the initial algebra $\Lambda In\Lambda$ for the signature Λ such that

$$\Lambda_{\{\}} = \{I\}, \qquad \Lambda_{\{1\}} = F, \qquad \Lambda_{\{1,2\}} = B.$$

This initial algebra is isomorphic to CT_Σ in Section 5.2, part I, of Reference 2. As a concrete syntax we write

1	for	$\Lambda In\Lambda_1$
f; x	for	$\Lambda In\Lambda_f(x)$
$b \to x_1,\ x_2$	for	$\Lambda In\Lambda_b(x_1,\ x_2)$.

This notation has been chosen so that general and linear flow diagrams with the same concrete representation should have the same meaning. Intuitively, applying this relationship to the forms I, f; x, and $b \to x_1, x_2$ determines the semantic equations for linear flow diagrams.

Thus direct semantics is provided by the semantic function $\mu_{\Lambda H} \in In\Lambda \to H$ such that

$$\mu_{\Lambda H}(I) = J_S$$

$$\mu_{\Lambda H}(f; x) = \mu_{\Lambda H}(x) * \mathcal{U}(f)$$

$$\mu_{\Lambda H}(b \to x_1, x_2) = \lambda s.\ cond_{S_\perp}(\mathcal{B}(b,s), \mu_{\Lambda H}(x_1,s), \mu_{\Lambda H}(x_2,s))$$

while continuation semantics is provided by $\mu_{\Lambda W} \in In\Lambda \to W$ such that:

$$\mu_{\Lambda W}(I) = I_C$$

$$\mu_{\Lambda W}(f; x) = \mathcal{U}(f) \cdot \mu_{\Lambda W}(x)$$

$$\mu_{\Lambda W}(b \to x_1, x_2) = \lambda c.\ \lambda s.\ cond_0(\mathcal{B}(b,s), \mu_{\Lambda W}(x_1,c,s), \mu_{\Lambda W}(x_2,c,s))$$

Decision Table Diagrams

There are a variety of equivalences for conditional branching operations. For example,

$$(b_1 \to (b_2 \to x_{11}, x_{12}), (b_2 \to x_{21}, x_{22}))$$
$$= (b_2 \to (b_1 \to x_{11}, x_{21}), (b_1 \to x_{12}, x_{22}))$$

or

$$(b_1 \to (b_2 \to x_{11}, x_{12}), (b_1 \to x_{21}, x_{22}))$$
$$= (b_1 \to (b_2 \to x_{11}, x_{12}), x_{22}) .$$

As a step towards eliminating this kind of redundancy, we will replace the set B of binary branching operations by a single many-way branching operation D, which is an idealization of the well-known programming concept of a decision table.

Suppose for a moment that $B = \{b_1, b_2\}$ has only two members. Then either side of the first equivalence given above can be replaced by

$$D\left[\begin{array}{c|ccc} b_1 \backslash b_2 & \underset{\sim}{\text{true}} & \underset{\sim}{\text{false}} & \bot \\ \hline \underset{\sim}{\text{true}} & x_{11} & x_{12} & \bot \\ \underset{\sim}{\text{false}} & x_{21} & x_{22} & \bot \\ \bot & \bot & \bot & \bot \end{array}\right].$$

Similarly, either side of the second equivalence can be replaced by

$$D\left[\begin{array}{c|ccc} b_1 \backslash b_2 & \underset{\sim}{\text{true}} & \underset{\sim}{\text{false}} & \bot \\ \hline \underset{\sim}{\text{true}} & x_{11} & x_{12} & \bot \\ \underset{\sim}{\text{false}} & x_{22} & x_{22} & x_{22} \\ \bot & \bot & \bot & \bot \end{array}\right].$$

Essentially, the decision $D[\underset{\sim}{x}]$ means "Produce a list t of the current value of all Boolean expressions in B, and then execute the table entry $\underset{\sim}{x}(t)$."

The "list" t is really a function in the domain $T = B \to Bool_{\bot}$. Thus if table entries belong to the set InΓ (which is going to be the language of decision table diagrams), then the decision table $\underset{\sim}{x}$ itself will be a function from T to InΓ. But only some of these functions are reasonable.

In the evaluation of any compound conditional branching operation, for a particular value of b_1, either b_2 will be evaluated, so that the nontermination of b_2 will imply the nontermination of the entire branching operation, or b_2 will not be evaluated, so that the outcome will be independent of b_2. This is reflected by the fact that every row in a decision table has either the form x y ⊥ or the form x x x. A similar rule holds for columns. As a consequence, decision tables are always monotonic functions from T to InΓ.

The generalization to arbitrary finite B is straightforward. More surprisingly, we can even permit B to be infinite. The only qualification, which is typical of the lattice-theoretic approach, is that decision tables are required to be continuous, rather than merely monotonic, functions from T to $In\Gamma$. Thus the decision operator D accepts operands which belong to $T \to In\Gamma$, i.e., it is an operator with rank T. Although such an operator goes beyond the framework of conventional algebra, it is encompassed by our generalization to operators whose ranks are arbitrary predomains.

Of course, an infinite decision table cannot be explicitly tabulated, but it can still be described by functional notation. For example, the decision table diagrams given above, when generalized to an arbitrary B containing b_1 and b_2, can be represented by

$$D[\lambda t.\ cond_{In\Gamma}(t(b_1),\ cond_{In\Gamma}(t(b_2),\ x_{11},\ x_{12}),\ cond_{In\Gamma}(t(b_2),\ x_{21},\ x_{22}))]$$

and

$$D[\lambda t.\ cond_{In\Gamma}(t(b_1),\ cond_{In\Gamma}(t(b_2),\ x_{11},\ x_{12}),\ x_{22})]$$

However, such expressions are not in themselves decision table diagrams, but only indirect and nonunique representations of such diagrams.

With this movitation, we can give a precise definition. The language of <u>decision table diagrams</u> is the initial algebra $\Gamma In\Gamma$ for the signature Γ such that

$$\Gamma_{\{\}} = \{I\}, \qquad \Gamma_{\{1\}} = F, \qquad \Gamma_T = \{D\}\ .$$

As a concrete syntax, we write

I	for	$\Gamma In\Gamma_I$
f; x	for	$\Gamma In\Gamma_f(x)$
D[x]	for	$\Gamma In\Gamma_D(x)$

Direct semantics is provided by the semantic function $\mu_{\Gamma H} \in In\Gamma \to H$ such that

$$\mu_{\Gamma H}(I) = J_S$$
$$\mu_{\Gamma H}(f; x) = \mu_{\Gamma H}(x) * \mathcal{F}(f)$$
$$\mu_{\Gamma H}(D[\underset{\sim}{x}]) = \lambda s.\ \mu_{\Gamma H}(\underset{\sim}{x}(\lambda b.\mathcal{B}(b,s)),\ s)$$

In the last equation, the function $\lambda b.\mathcal{B}(b,s) \in T$ denotes the "list" of the values of all Boolean expressions in the state s.

For continuation semantics, we have the semantic function $\mu_{\Gamma W} \in In\Gamma \to W$ such that

$$\mu_{\Gamma W}(I) = I_C$$
$$\mu_{\Gamma W}(f; x) = \mathcal{G}(f) \cdot \mu_{\Gamma W}(x)$$
$$\mu_{\Gamma W}(D[\underset{\sim}{x}]) = \lambda c.\ \lambda s.\ \mu_{\Gamma W}(\underset{\sim}{x}(\lambda b.\mathcal{B}(b,\ s)),\ c,\ s)$$

Admittedly, we are stretching a point in calling decision table diagrams a language. They are even further than flow diagrams from the conventional finitary concept of language. But they still have the essential linguistic characteristic of being uninterpreted: They make no commitment to a choice of the set of states, nor to the meaning of primitive instructions or Boolean expressions, nor even to the choice between direct and continuation semantics.

Processes

There are two important equivalences for decision table diagrams. The first,

$$D[\lambda t.\ x] = x\ ,$$

shows that a decision is redundant if all of its table entries are the same. The second,

$$D[\lambda t.\ D[\lambda t'.\ g(t,\ t')]] = D[\lambda t.\ g(t,\ t)]$$

where $g \in T \to (T \to In\Gamma)$, shows that, when the entries of a decision table are

themselves decisions, the inner decisions must "go the same way" as the outer one, since there is no intervening primitive instruction which might change the state of the computation.

To eliminate this kind of redundancy, we will define a further language in which decisions and primitive instructions are required to alternate. More precisely, a decision table entry will always have the form I, or $\perp$, or f; D[x] where x is a decision table.

To obtain an algebraic formulation, we regard f;D[x] as the application of a T-ary operator, named by f, to the operand x. Then table entries are the initial algebra $\Delta\text{In}\Delta$ for the signature Δ such that

$$\Delta_{\{\}} = \{I\}, \qquad \Delta_T = F \ .$$

As a concrete syntax, we write

$$\begin{array}{lll} I & \text{for} & \Delta\text{In}\Delta_I \\ f;\ D[x] & \text{for} & \Delta\text{In}\Delta_f(x) \ , \end{array}$$

which suggests the relationship between our new language and the language of decision table diagrams.

If $\text{In}\Delta$ is the domain of table entries, then decision tables themselves belong to the domain $T \to \text{In}\Delta$, which we will call Z. From equation (1) in the proof of Theorem 1, $\text{In}\Delta$ satisfies the domain isomorphism

$$\begin{aligned} \text{In}\Delta &\approx \sum_{\sigma \in \{I\} \cup F} (\text{rank}(\sigma) \to \text{In}\Delta) \\ &= \sum_{\sigma \in \{I\} \cup F} \underset{\sim}{\text{if}}\ \sigma = I\ \underset{\sim}{\text{then}}\ (\{\} \to \text{In}\Delta)\ \underset{\sim}{\text{else}}\ (T \to \text{In}\Delta) \\ &= \sum_{\sigma \in \{I\} \cup F} \underset{\sim}{\text{if}}\ \sigma = I\ \underset{\sim}{\text{then}}\ \{\cdot\}\ \underset{\sim}{\text{else}}\ Z \end{aligned}$$

where $\{\cdot\}$ denotes a one-element domain. But the right side is easily seen to be isomorphic to $\{\cdot\} + F \times Z$. Thus Z and $\text{In}\Delta$ satisfy

$$Z = T \to \text{In}\Delta \ , \qquad \text{In}\Delta \approx \{\cdot\} + F \times Z \ .$$

These "domain equations" suggest a close connection with the concept of processes developed by Milner[6] and Bekic,[10] which in fact inspired the language described here. To emphasize this connection we will henceforth call Z the domain of processes and InΔ the domain of process components. However, it should be noted that the processes of Milner and Bekic are less syntactic than ours, and are specifically oriented to problems of concurrent processing which are not considered here.

Intuitively, the process z means "For the current state s, compute a list $t = \lambda b.\mathcal{B}(b,s)$ of values of the Boolean expressions, and then execute the process component z(t)." The process component I means "Do nothing," while the process component f; D[z] means "Execute f and then execute the process z."

This intuition is captured in direct semantics by the semantic functions $\delta_{\Gamma H} \in Z \to H$ and $\mu_{\Delta H} \in In\Delta \to H$ such that

$$\delta_{\Gamma H}(z) = \lambda s.\ \mu_{\Delta H}(z(\lambda b.\mathcal{B}(b,s)),\ s)$$
$$\mu_{\Delta H}(I) = J_S$$
$$\mu_{\Delta H}(f;\ D[z]) = \delta_{\Gamma H}(z) * \mathcal{F}(f)$$

For continuation semantics, the semantic functions are $\delta_{\Gamma W} \in Z \to W$ and $\mu_{\Delta W} \in In\Delta \to W$ such that

$$\delta_{\Gamma W}(z) = \lambda c.\ \lambda s.\ \mu_{\Delta W}(z(\lambda b.\mathcal{B}(b,s)),\ c,\ s)$$
$$\mu_{\Delta W}(I) = I_C$$
$$\mu_{\Delta W}(f;\ D[z]) = \mathcal{G}(f) \cdot \delta_{\Gamma W}(z)$$

For either set of semantic equations, the substitution of the first equation into the third gives a pair of homomorphic equations for $\mu_{\Delta H}$ or $\mu_{\Delta W}$. We have introduced the subsidiary functions $\delta_{\Gamma H}$ and $\delta_{\Gamma W}$ (whose names will become meaningful later) to treat the domains Z and InΔ more symmetrically.

Nevertheless, Z, unlike InΔ, is not the carrier of an initial algebra. This anomaly is the price of avoiding many-sorted algebras. One could define Z and InΔ as the carriers of a two-sorted initial algebra, with $\delta_{\Gamma H}$ and $\mu_{\Delta H}$ (or $\delta_{\Gamma W}$ and $\mu_{\Delta W}$) as the components of a two-sorted homomorphism.

An Algebraic Treatment of Decision Table Diagrams

From an algebraic viewpoint, the eight language definitions we have given are tantamount to a specification of the following target algebras:

Language	Direct Semantics	Continuation Semantics
General flow diagrams	ΩH	ΩW
Linear flow diagrams	ΛH	ΛW
Decision table diagrams	ΓH	ΓW
Process components	ΔH	ΔW

Our main task is to define these target algebras more directly, and to investigate relationships among them. As each target algebra is defined, it will become apparent that the corresponding language definition is a display of the equations for the homomorphism into the target algebra from the initial algebra with the same signature.

We will begin with decision table diagrams, then move "forward" to processes, and then move "backwards" to linear, and finally general flow diagrams.

Decision table digarams are the initial algebra with the signature Γ such that $\Gamma_{\{\}} = \{I\}$, $\Gamma_{\{1\}} = F$, and $\Gamma_T = \{D\}$, where $T = B \to Bool_\perp$. To describe their direct semantics, let S be some predomain of states, $H = S \to S_\perp$, $\mathcal{J} \in F \to H$, and $\mathcal{B} \in B \to (S \to Bool_\perp)$. Then the target algebra ΓH is the Γ-algebra with carrier H such that

$$\Gamma H_I = J_S$$

$$\Gamma H_f(h) = h^* \mathcal{J}(f)$$

$$\Gamma H_D(h) = \lambda s.\ h(\lambda b.\ \mathcal{B}(b, s), s) \ .$$

For continuation semantics, let S be some predomain of states, O be some domain of outputs, $C = S \to O$, $W = C \to C$, $\mathcal{G} \in F \to W$, and $\mathcal{B} \in B \to (S \to Bool_\perp)$. Then the target algebra ΓW is the Γ-algebra with carrier W such that

$$\Gamma W_I = I_C$$

$$\Gamma W_f(w) = \mathcal{G}(f) \cdot w$$

$$\Gamma W_D(w) = \lambda c.\ \lambda s.\ w(\lambda b.\ \mathcal{B}(b, s), c, s) \ .$$

The previously given semantic equations for decision table diagrams are simply the homomorphic equations for the unique homomorphisms $\mu_{\Gamma H} \in \Gamma In\Gamma \to \Gamma H$ and $\mu_{\Gamma W} \in \Gamma In\Gamma \to \Gamma W$.

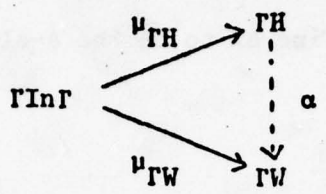
We can now establish the fundamental relationship between direct and continuation semantics. First note that ΓH is actually a family of Γ-algebras depending upon the triple $\langle S, \mathcal{F}, \mathcal{B} \rangle$, which we will call a direct interpretation. Similarly, ΓW is actually a family depending upon the quadruple $\langle S, O, \mathcal{G}, \mathcal{B} \rangle$, which we will call a continuation interpretation. (This kind of dependence will hold for all of our target algebras describing direct or continuation semantics.)

Consider a direct interpretation $\langle S, \mathcal{F}, \mathcal{B} \rangle$ and a continuation interpretation $\langle S, O, \mathcal{G}, \mathcal{B} \rangle$ with the same S and $\mathcal{B}$. Let $\alpha \in H \to W$ be the function such that $\alpha(h) = \lambda c.\ c^*h$. If $\mathcal{G} = \alpha \cdot \mathcal{F}$, then the continuation interpretation is said to be an image of the direct interpretation. If, in addition, $O = S_\perp$, then it is an exact image. Each direct interpretation obviously has many images, one of which is exact. However, there are continuation interpretations (where primitive instructions have "abnormal" meanings) which are not the image of any direct interpretation. Then:

> Proposition 1 If the continuation interpretation is an image of the direct interpretation, then is a homomorphism from ΓH to ΓW. If the image is exact, then a left inverse for α is provided by the function $\beta \in W \to H$ such that $\beta(w) = w(J_S)$.

Proof: The reader may verify that α satisfies the equations for a homomorphism from ΓH to ΓW, and that it is strict and continuous. In the exact case, $\beta(\alpha(h)) = \alpha(h)(J_S) = J_S{}^*h = h$. $\square$

Thus we have the following diagram of homomorphisms:



(I)

where α occurs only if the continuation interpretation is an image of the direct interpretation. Since $\Gamma In\Gamma$ is an initial algebra, which implies $\alpha \cdot \mu_{\Gamma H} = \mu_{\Gamma W}$, this diagram commutes, i.e., all paths with the same origin and destination denote equal compositions of functions.

In other words, when the continuation interpretation is an image of the direct interpretation, α maps the direct meaning of any decision table diagram into its continuation meaning. Moreover, when the image is exact, β maps the continuation meaning back into the direct meaning.

Processes

Let Δ be the signature such that $\Delta_{\{\}} = \{I\}$ and $\Delta_T = F$. Then the domain of process components is $In\Delta$, and the domain of processes is $Z = T \rightarrow In\Delta$. Not only do we want to define the semantics of these entities, but also to connect this semantics with the semantics of decision table diagrams. For this purpose, we will assume that the semantics of decision table diagrams is given by an arbitrary target algebra ΓX (of which ΓH and ΓW are instances), and we will derive the semantics of processes and their components from ΓX.

Intuitively, in a Γ-algebra, ΓX_I means "Do nothing," $\Gamma X_f(x)$ means "Execute f and then execute x," and $\Gamma X_D(\underset{\sim}{x})$ means "Compute a table t of the current values of Boolean expressions and then exectue $\underset{\sim}{x}(t)$." In a Δ-algebra, ΔX_I means "Do nothing," and $\Delta X_f(x)$ means "Execute f, compute a table t of the current values of Boolean expressions, and then execute $\underset{\sim}{x}(t)$." This suggests that, for any Γ-algebra ΓX, we define ΔX to be the Δ-algebra with the same carrier such that

$$\Delta X_I = \Gamma X_I$$
$$\Delta X_f(\underset{\sim}{x}) = \Gamma X_f(\Gamma X_D(\underset{\sim}{x})) \ .$$

Now suppose that $z \in Z$ is a process. Then the meaning of a process component $z(t)$ will be $\mu_{\Delta X}(z(t))$. But the meaning of z itself is "Compute a table t of the current values of Boolean expressions and then execute the meaning of $z(t)$," or in other words, ΓX_D of the function of t which gives the meaning of $z(t)$. Thus processes are mapped into their meanings by the function $\delta_{\Gamma X} \in Z \to X$ such that

$$\delta_{\Gamma X}(z) = \Gamma X_D(\lambda t.\ \mu_{\Delta X}(z(t))) = \Gamma X_D(\mu_{\Delta X} \cdot z) \ .$$

The previously given semantic equations for processes are simply assertions that the direct and continuation semantics of processes are given by $\delta_{\Gamma H}$ and $\delta_{\Gamma W}$, and the direct and continuation semantic of process components are given by $\mu_{\Delta H}$ and $\mu_{\Delta W}$.

The δ-functions bear the following relationship to homomorphisms between Γ-algebras:

Proposition 2 If $\rho \in \Gamma X \to \Gamma X'$, then $\rho \cdot \delta_{\Gamma X} = \delta_{\Gamma X'} \cdot$

Proof: Suppose $\rho \in \Gamma X \to \Gamma X'$. It is easily seen that this implies $\rho \in \Delta X \to \Delta X'$ (which is not surprising, since the definition of ΔX in terms of ΓX is an instance of the standard notion of a "derived algebra"). But then

$\rho \cdot \mu_{\Delta X} \in \Delta In\Delta \to \Delta X'$ must be the unique homomorphism $\mu_{\Delta X'}$. Thus $\rho(\delta_{\Gamma X}(z))$ $= \rho(\Gamma X_D(\mu_{\Delta X} \cdot z)) = \Gamma X'_D(\rho \cdot \mu_{\Delta X} \cdot z) = \Gamma X'_D(\mu_{\Delta X'} \cdot z) = \delta_{\Gamma X'}$. □

Informally, we motivated the definition of processes by suggesting that they were a kind of canonical form for decision table diagrams. If this is true, then it should be possible to translate decision table diagrams into processes. By regarding this translation as a kind of meaning, we can define it algebraically: we will define a target algebra ΓZ such that decision table diagrams are translated into processes by the unique homomorphism $\mu_{\Gamma Z} \in \Gamma In\Gamma \to \Gamma Z$. It turns out that the right definition of ΓZ is the Γ-algebra with carrier Z such that

$$\Gamma Z_I = \lambda t.\ \Delta In\Delta_I$$
$$\Gamma Z_f(z) = \lambda t.\ \Delta In\Delta_f(z)$$
$$\Gamma Z_D(\underset{\sim}{z}) = \lambda t.\ \underset{\sim}{z}(t)(t) \qquad (\text{where } \underset{\sim}{z} \in T \to Z = T \to (T \to In\Delta)).$$

This definition leads to the following relationship between ΓZ and the δ-functions:

Proposition 3 (1) ΓZ is a key algebra. If it exists, the unique homomorphism from ΓZ to ΓX is $\delta_{\Gamma X}$. (2) $\delta_{\Gamma In\Gamma}$ is a right inverse for $\mu_{\Gamma Z}$.

Proof: The definition of ΓZ gives rise to the derived algebra ΓZ. By checking the appropriate homomorphic equations, the reader may verify that $\lambda x.\ \lambda t.\ x$ is a homomorphism from $\Delta In\Delta$ to ΔZ, and is therefore equal to the unique homomorphism $\mu_{\Delta Z}$. This implies that $\delta_{\Gamma Z}$ is the identity function for Z, since $\delta_{\Gamma Z}(z) = \Gamma Z_D(\mu_{\Delta Z} \cdot z) = \lambda t.\ (\mu_{\Delta Z} \cdot z)(t)(t) = \lambda t.\ \mu_{\Delta Z}(z(t))(t) = \lambda t.\ z(t)$ $= z$.

To establish (1), let ρ be any homomorphism from ΓZ to ΓX. Then $\rho = \rho \cdot \delta_{\Gamma Z}$, and by Proposition 2, $\rho \cdot \delta_{\Gamma Z} = \delta_{\Gamma X}$. To establish (2), take ρ in Proposition 2 to be $\mu_{\Gamma Z} \in \Gamma In\Gamma \to \Gamma Z$. Then $\mu_{\Gamma Z} \cdot \delta_{\Gamma In\Gamma} = \delta_{\Gamma Z} = I_Z$. □

Although ΓZ is a key algebra, it is not initial, since it is not isomorphic to $\Gamma In\Gamma$. In other words, there are Γ-algebras for which there is no homomorphism from ΓZ. However, we are really only interested in target algebras in which D behaves like a decision operation. This behavior is characterized by the following <u>decision laws</u>, which are the algebraic formulation of the equivalences for D-operations given earlier:

(1) $(\forall x \in X)\ \Gamma X_D(\lambda t.\ x) = x$

(2) $(\forall g \in T \to (T \to X))$

$$\Gamma X_D(\lambda t.\ \Gamma X_D(\lambda t'.\ g(t,t'))) = \Gamma X_D(\lambda t.\ g(t,t))$$

The following proposition shows that ΓZ is initial in the restricted class of Γ-algebras which obey the decision laws. The reader may verify that this class includes ΓH, ΓW, and ΓZ, but not $\Gamma In\Gamma$.

<u>Proposition 4</u> If ΓX obeys the decision laws, then $\delta_{\Gamma Z} \in \Gamma Z \to \Gamma X$.

Proof: The necessary homomorphic equations are established by:

$$\delta_{\Gamma X}(\Gamma Z_I) = \Gamma X_D(\mu_{\Delta X} \cdot (\lambda t.\ \Delta In\Delta_I)) = \Gamma X_D(\lambda t.\ \mu_{\Delta X}(\Delta In\Delta_I))$$
$$= \mu_{\Delta X}(\Delta In\Delta_I) = \Delta X_I = \Gamma X_I$$

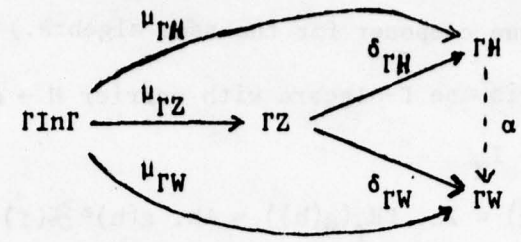
$$\delta_{\Gamma X}(\Gamma Z_f(z)) = \Gamma X_D(\mu_{\Delta X} \cdot (\lambda t.\ \Delta In\Delta_f(z)))$$
$$= \Gamma X_D(\lambda t.\ \mu_{\Delta X}(\Delta In\Delta_f(z))) = \mu_{\Delta X}(\Delta In\Delta_f(z)) = \Delta X_f(\mu_{\Delta X} \cdot z)$$
$$= \Gamma X_f(\Gamma X_D(\mu_{\Delta X} \cdot z)) = \Gamma X_f(\delta_{\Gamma X}(z))$$

$$\delta_{\Gamma X}(\Gamma Z_D(\underset{\sim}{z})) = \Gamma X_D(\mu_{\Delta X} \cdot (\lambda t.\ \underset{\sim}{z}(t)(t))) = \Gamma X_D(\lambda t.\ \mu_{\Delta X}(\underset{\sim}{z}(t)(t)))$$
$$= \Gamma X_D(\lambda t.\ \Gamma X_D(\lambda t'.\ \mu_{\Delta X}(\underset{\sim}{z}(t)(t')))) = \Gamma X_D(\lambda t.\ \Gamma X_D(\mu_{\Delta X} \cdot \underset{\sim}{z}(t)))$$
$$= \Gamma X_D(\lambda t.\ \delta_{\Gamma X}(\underset{\sim}{z}(t))) = \Gamma X_D(\delta_{\Gamma X} \cdot \underset{\sim}{z})$$

Strictness is established by:

$$\delta_{\Gamma X}(\perp) = \Gamma X_D(\mu_{\Delta X} \cdot \perp) = \Gamma X_D(\lambda t.\ \mu_{\Delta X}(\perp)) = \mu_{\Delta X}(\perp) = \perp \quad \square$$

At this stage, we have the following diagram of homomorphisms:



(II)

Since ΓInΓ and ΓZ are both key algebras, the diagram commutes.

Composers and Monoids

To treat linear and general flow diagrams, we will derive Λ-algebras from the Γ-algebras, and then derive Ω-algebras from the Λ-algebras. However, in the final stage, in order to define the semicolon operation for Ω-algebras, we will need appropriate composition functions, or composers, for each of our algebras. We will introduce these composers as we go along, and show at each stage that the homomorphisms we use remain valid when each algebra is augmented by adding its composer as an additional binary operation.

In general, when ΣX is a Σ-algebra with carrier X, and $\Sigma_{\{\}}$ contains the constant I, we write FnΣX for the Σ-algebra with carrier X → X such that

$$\mathrm{Fn}\Sigma X_I = I_X$$

For $\sigma \in \Sigma_S$, $\sigma \neq I$, and $\underset{\sim}{g} \in S \to (X \to X)$,

$$\mathrm{Fn}\Sigma X_\sigma(\underset{\sim}{g}) = \lambda x.\ \Sigma X_\sigma(\lambda s.\ \underset{\sim}{g}(s)(x)) \quad .$$

When S = {1, ... , n}, the conventional form of the last equation is:

$$\mathrm{Fn}\Sigma X_\sigma(g_1, \ldots, g_n) = \lambda x.\ \Sigma X_\sigma(g_1(x), \ldots, g_n(x)) \ .$$

A homomorphism η ∈ ΣX → FnΣX is said to be a composer for ΣX.

For any signature Σ, we write Σ; to indicate the signature obtained from Σ by adding ; as a binary operator. When η is a composer for ΣX, we write Σ;X to denote the algebra obtained from ΣX by interpreting ; as the binary operation

$\Sigma^{;}\Sigma_{;} = \eta$. (There is a potential ambiguity here which we will avoid by never specifying more than one composer for the same algebra.)

For example $Fn\Gamma H$ is the Γ-algebra with carrier $H \to H$ such that

$$Fn\Gamma H_I = I_H$$

$$Fn\Gamma H_f(g) = \lambda h.\ \Gamma H_f(g(h)) = \lambda h.\ g(h)*\mathcal{H}(f)$$

$$Fn\Gamma H_D(\underset{\sim}{g}) = \lambda h.\ \Gamma H_D(\lambda t.\ \underset{\sim}{g}(t)(h))$$

$$= \lambda h.\ \lambda s.\ \underset{\sim}{g}(\lambda b.\mathcal{B}(b,s),\ h,\ s)\ .$$

It is easily seen that the function $\lambda h_1.\ \lambda h_2.\ h_2*h_1$ is a homomorphism from ΓH to $Fn\Gamma H$, and is therefore a composer for ΓH. Thus $\Gamma^{;}H$ is obtained from ΓH by interpreting the semicolon as $\Gamma^{;}H_{;}(h_1, h_2) = h_2*h_1$.

This construction has the following general properties:

Theorem 2 If $I \in \Sigma_{\{\}}$ and ΣX is a key Σ-algebra with a composer η, then $\Sigma^{;}X$ is a monoid, i.e.,

(1) $(\forall x \in X)\ \eta(\Sigma X_I, x) = x$

(2) $(\forall x \in X)\ \eta(x, \Sigma X_I) = x$

(3) $(\forall x, y, z \in X)\ \eta(\eta(x, y), z) = \eta(x, \eta(y, z))$.

If in addition, $\Sigma X'$ is a Σ-algebra with a composer and $\rho \in \Sigma X \to \Sigma X'$, then $\rho \in \Sigma^{;}X \to \Sigma^{;}X'$.

Proof: For a Σ-algebra ΣX and $x \in X$, we write $\Sigma X[x]$ to denote the algebra obtained from ΣX by reinterpreting I as the constant $\Sigma X[x]_I = x$. It is evident that $\rho \in \Sigma X \to \Sigma X'$ implies $\rho \in \Sigma X[x] \to \Sigma X'[\rho(x)]$. Also, the definition of $Fn\Sigma X$ implies that $\lambda g.\ g(x) \in Fn\Sigma X \to \Sigma X[x]$.

Now suppose ΣX is a key algebra with the composer $\eta \in \Sigma X \to Fn\Sigma X$. Then:

(1) $\eta(\Sigma X_I, x) = Fn\Sigma X_I(x) = x$.

(2) The assumption that ΣX is a key algebra implies that the diagram

$$I_X \circlearrowright \Sigma X \underset{\lambda g.\ g(\Sigma X_I)}{\overset{\eta}{\rightleftarrows}} Fn\Sigma X$$

of homomorphisms commutes. Applying both compositions to x gives $\eta(x, \Sigma X_I) = x$.

(3) The following diagram also commutes:

$$\begin{array}{ccccc}
 & & \overset{\eta}{} & & \overset{\lambda g.\ g(y)}{} \\
\Sigma X & \longrightarrow & Fn\Sigma X & \longrightarrow & \Sigma X[y] \\
\Big\downarrow{\scriptstyle \eta} & & & & \Big\downarrow{\scriptstyle \eta} \\
 & & & & Fn\Sigma X[\eta(y)] \\
 & & \overset{\lambda g.\ g(\eta(y)(z))}{} & & \Big\downarrow{\scriptstyle \lambda g.\ g(z)} \\
Fn\Sigma X & & \longrightarrow & & \Sigma X[z][\eta(y)(z)] \\
 & & & & = \Sigma X[\eta(y)(z)]
\end{array}$$

Applying both compositions to x gives $\eta(\eta(x, y), z) = \eta(x, \eta(y, z))$.

If $\Sigma X'$ has the composer η' and $\rho \in \Sigma X \to \Sigma X'$, then the following diagram also commutes:

$$\begin{array}{ccccc}
 & & \overset{\eta}{} & & \overset{\lambda g.\ g(y)}{} \\
\Sigma X & \longrightarrow & Fn\Sigma X & \longrightarrow & \Sigma X[y] \\
\Big\downarrow{\scriptstyle \rho} & & & & \Big\downarrow{\scriptstyle \rho} \\
 & \overset{\eta'}{} & & \overset{\lambda g'.\ g'(\rho(y))}{} & \\
\Sigma X' & \longrightarrow & Fn\Sigma X' & \longrightarrow & \Sigma X'[\rho(y)]
\end{array}$$

Applying both compositions to x gives $\rho(\eta(x, y)) = \eta'(\rho(x), \rho(y))$, which is the extra homomorphic equation needed to show $\rho \in \Sigma^i X \to \Sigma^i X'$. □

Our immediate goal is to extend Diagram II to the signature Γ'. Suppose that we can provide a composer for each of the algebras in this diagram, and that we can show that $\alpha \in \Gamma'H \to \Gamma'W$. Then since every other homomorphism in (II) has the form $\rho \in \Gamma X \to \Gamma X'$ where ΓX is a key algebra (either $\Gamma In\Gamma$ or ΓZ), Theorem 2 gives $\rho \in \Gamma'X \to \Gamma'X'$. Thus (II) will remain a diagram of homomorphisms if we change the signature of each algebra to Γ', i.e., if we add the appropriate composer to each algebra as a binary operation. Of course, the diagram will continue to commute since the functions remain unchanged.

By deriving the appropriate algebras, verifying the resulting homomorphic equations, and checking strictness, one can show that

> Proposition 5 (1) $\lambda h_1.\ \lambda h_2.\ h_2 * h_1$ is a composer for ΓH. (2) $\lambda w_1.\ \lambda w_2.\ w_1 \cdot w_2$ is a composer for ΓW. (3) If the continuation interpretation is an image of the direct interpretation, then $\alpha \in \Gamma'H \to \Gamma'W$.

Moreover, a composer for $\Gamma In\Gamma$ is automatically provided by initiality. In general,

> Proposition 6 For any signature Σ, $\mu_{Fn\Sigma In\Sigma}$ is the (unique) composer for $\Sigma In\Sigma$.

To complete our chain of reasoning, we must find a composer for ΓZ. To do so, we use an analogue of the idea of an Herbrand interpretation: We define a particular continuation semantics in which continuations are processes, and the meaning of a process is a function which forms the composition of that process with another.

Let ΓW^o be the special case of ΓW for the continuation interpretation in which $S = T$, $O = In\Delta$, $\mathcal{J} = \lambda f.\ \Gamma Z_f$ and $\mathcal{B} = \lambda b.\ \lambda t.\ t(b)$, so that $C = Z$, $W = Z \to Z$, and

$$\Gamma W^{c}_{I} = I_{Z}$$

$$\Gamma W^{o}_{f}(w) = \Gamma Z_{f} \cdot w$$

$$\Gamma W^{o}_{D}(\underset{\sim}{w}) = \lambda z.\ \Gamma Z_{D}(\lambda t.\ \underset{\sim}{w}(t, z)) \quad .$$

Then ΓW^{o} is identical with $Fn\Gamma Z$, so that

Proposition 7 $\delta_{\Gamma W^{o}}$ is the (unique) composer for ΓZ.

A side benefit is the right identity law for the monoid $\Gamma^{\cdot}Z$, $\delta_{\Gamma W^{o}}(z)(\Gamma Z_{I}) = z$, which implies

Proposition 8 $\lambda w.\ w(\Gamma Z_{I}) \in (Z \to Z) \to Z$ is a left inverse for $\delta_{\Gamma W^{o}}$.

Thus processes are canonical for continuation semantics, since ΓW^{o} is a continuation semantics which always gives distinct meanings to distinct processes.

In contrast, there is no direct semantics with this property. For example, the distinct processes $\perp$ and $\Gamma Z_{f}(\perp)$ both mean $\perp$ in any direct semantics. Thus direct semantics induces a coarser relation of equivalence for processes (or other languages) than continuation semantics. This situation coincides with the author's intuition about the difference between direct and continuation semantics. It is one of the main reasons for not using a complete-lattice formalism; we have not been able to find a simple complete-lattice formulation of direct semantics such that $\perp$ and $\Gamma Z_{f}(\perp)$ have the same meaning in all direct and continuation semantics.

Linear Flow Diagrams

Linear flow diagrams are the initial algebra with the signature Λ such that $\Lambda_{\{\}} = \{I\}$, $\Lambda_{\{1\}} = F$, and $\Lambda_{\{1,2\}} = B$. Intuitively, in a Λ-algebra I and each f have the same meaning as in the corresponding Γ-algebra, while

$\Lambda X_b(x_1, x_2)$ means "Compute the current value of b and then execute x_1 if this value is true or x_2 if it is false." Thus, for any Γ-algebra ΓX, we define ΛX to be the Λ-algebra with the same carrier such that

$$\Lambda X_I = \Gamma X_I$$

$$\Lambda X_f = \Gamma X_f$$

$$\Lambda X_b(x_1, x_2) = \Gamma X_D(\lambda t.\ cond_X(t(b), x_1, x_2)) .$$

Then:

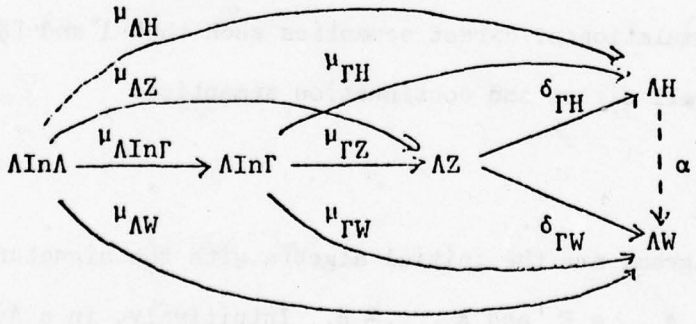
Proposition 9 (1) If $\rho \in \Gamma X \to \Gamma X'$ then $\rho \in \Lambda X \to \Lambda X'$.

(2) A composer for ΓX is also a composer for ΛX.

(3) If $\rho \in \Gamma^i X \to \Gamma^i X'$ then $\rho \in \Lambda^i X \to \Lambda^i X'$.

The tedious but straightforward proof is left to the reader. However, it should be noted that these results are more than an application of the standard notion of a derived algebra, since they depend upon properties of the conditional function and the strictness of homomorphisms.

As a consequence (II) remains a diagram of homomorphisms if we change the signature of all algebras to either Λ or Λ^i. For Λ, we can add the initial algebra $\Lambda In\Lambda$ and its unique homomorphisms to the other algebras:



(III)

But Proposition 6 provides a composer for the added algebra, and Theorem 2 insures that the added homomorphisms extend to Λ^i. Thus (III) remains a diagram of homomorphisms if we change each signature to Λ^i.

There is no right inverse for $\mu_{\Lambda In\Gamma}$, since there are decision table diagrams which cannot be mapped into any flow diagram with the same meaning in all semantics. A simple example is the decision table diagram

$$D[\lambda t.\ (f_1;\ cond_{In\Gamma}(t(b),\ (f_2;\ I),\ (f_3;\ I)))] \ .$$

To execute this diagram, one must save a copy of the initial state during the execution of f_1, and then evaluate b in the saved state to determine whether f_2 or f_3 is to be executed next. One cannot evaluate b before executing f_1 since b may be nonterminating and f_1 may be an abnormal instruction.

A more spectular example, for which the initial state must be saved indefinitely, is the decision table diagram $D[\lambda t.\ \theta(0,\ t)]$, where $\theta \in Int \to (T \to In\Gamma)$ is the least solution of

$$\theta(n,\ t) = cond_{In\Gamma}(t(b_n),\ (f_1;\ \theta(n+1,t)),\ (f_2;\ \theta(n+1,t)))$$

and $\{b_0,\ b_1,\ \ldots\}$ is some enumeration of B.

The existence of such decision table diagrams is regretable, since their execution requires a kind of wholesale state-saving that is not in the spirit of imperative programming languages. It is an open question whether our development could be carried out with some more restricted notion of decision table diagram (and of process) such that every decision table diagram is equivalent to some flow diagram.

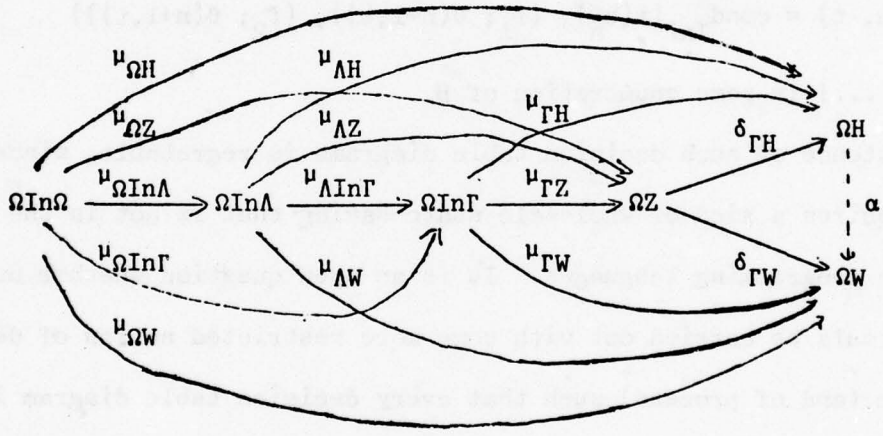
General Flow Diagrams

General flow diagrams are the initial algebra with the signature Ω such that $\Omega_{\{\}} = \{I\} \cup F$ and $\Omega_{\{1,2\}} = \{;\} \cup B$. For any Λ^i-algebra $\Lambda^i X$, we define ΩX to be the Ω-algebra with the same carrier such that

$$\Omega X_I = \Lambda' X_I$$
$$\Omega X_f = \Lambda' X_f(\Lambda' X_I)$$
$$\Omega X_; = \Lambda' X_;$$
$$\Omega X_b = \Lambda' X_b \ .$$

Note that Λ' is used, rather than Λ, to provide an interpretation of the semicolon operator - we are finally letting our composers perform publicly. This is a standard instance of a derived algebra, so that $\rho \in \Lambda' X \to \Lambda' X'$ implies $\rho \in \Omega X \to \Omega X'$.

Thus, since the functions in Diagram III are homomorphisms of Λ'-algebras, they are also homomorphisms of Ω-algebras. By adding the initial Ω-algebra and its unique homomorphisms to the other algebras, we obtain:



(IV)

Since $\Omega\text{In}\Omega$ is initial, the diagram continues to commute.

An implication is that $\mu_{\Omega\text{In}\Lambda}$ is a meaning-preserving function from general to linear flow diagrams, which is slightly surprising since, intuitively, general flow diagrams really seem to be more general than linear flow diagrams. In fact this is a valid intuition, but in a more subtle sense: Call an element of an algebra <u>equational</u> if it is the least solution of a finite set of finite

equations whose right sides are constructed from the operators of the algebra (as in Section 5.1 of Reference 2). Then there are equational elements of $\Omega In\Omega$, such as the least solution of

$$x = b \rightarrow ((f;\ x);\ g),\ 1$$

which are mapped by $\mu_{\Omega In\Lambda}$ into nonequational elements of $\Lambda In\Lambda$. Essentially, the equational elements of a Ω-algebra can express recursion, while those of a Λ-algebra can only express iteration.

Finally, we can "redefine" a Λ-algebra in terms of an Ω-algebra. For any Ω-algebra ΩX (actually, we will only be interested in $\Omega In\Omega$ and $\Omega In\Lambda$), let ΛX be the Λ-algebra with the same carrier such that

$$\Lambda X_I = \Omega X_I$$
$$\Lambda X_f(x) = \Omega X_;(\Omega X_f,\ x)$$
$$\Lambda X_b = \Omega X_b\ .$$

Again this is a standard instance of a derived algebra, so that $\rho \in \Omega X \rightarrow \Omega X'$ implies $\rho \in \Lambda X \rightarrow \Lambda X'$.

By applying this fact to $\mu_{\Omega In\Lambda}$ and adding the initial Λ-algebra and its unique homomorphism into $\Lambda In\Omega$, we get

$$\Lambda In\Lambda \xrightarrow{\mu_{\Lambda In\Omega}} \Lambda In\Omega \xrightarrow{\mu_{\Omega In\Lambda}} \Lambda In\Lambda'$$

where the algebra on the left is the initial Λ-algebra and the algebra on the right is obtained from $\Omega In\Lambda$ by the above definition. But in fact these two algebras are the same, as can be seen by comparing their operations. The only nontrivial case occurs for the operator f, where

$$\begin{aligned}
\Lambda In\Lambda'_f(x) &= \Omega In\Lambda_;(\Omega In\Lambda_f, x) \\
&= \Lambda' In\Lambda_;(\Lambda' In\Lambda_f(\Lambda' In\Lambda_I), x) \\
&= \mu_{Fn\Lambda In\Lambda}(\Lambda In\Lambda_f(\Lambda In\Lambda_I))(x) \\
&= Fn\Lambda In\Lambda_f(Fn\Lambda In\Lambda_I)(x) \\
&= (\lambda x.\ \Lambda In\Lambda_f(Fn\Lambda In\Lambda_I(x)))(x) \\
&= (\lambda x.\ \Lambda In\Lambda_f(I_{In\Lambda}(x)))(x) = \Lambda In\Lambda_f(x)\ .
\end{aligned}$$

Then since $\Lambda In\Lambda$ is initial, we have $\mu_{\Omega In\Lambda} \cdot \mu_{\Lambda In\Omega} = \mu_{\Lambda In\Lambda} = I_{In\Lambda}$, i.e.,

Proposition 10 $\mu_{\Lambda In\Omega}$ is a right inverse for $\mu_{\Omega In\Lambda}$.

(But not a left inverse, since $\mu_{\Omega In\Lambda}$ is a many-one function from general to linear flow diagrams.)

Conclusions

A variety of relationships among our languages and their semantics are specified by the commutativity of Diagram IV, along with Propositions 1, 3.2, 8, and 10. Most of this information can be summarized as follows:

(1) The following are continuous functions among general flow diagrams ($In\Omega$), linear flow diagrams ($In\Lambda$), decision table diagrams ($In\Gamma$), and processes (Z), which preserve meaning in any direct or continuation semantics:

$$In\Omega \underset{\mu_{\Lambda In\Omega}}{\overset{\mu_{\Omega In\Lambda}}{\rightleftarrows}} In\Lambda \xrightarrow{\mu_{\Lambda In\Gamma}} In\Gamma \underset{\delta_{\Gamma In\Gamma}}{\overset{\mu_{\Gamma Z}}{\rightleftarrows}} Z$$

However, there is no such meaning-preserving function from decision table diagrams to linear flow diagrams.

(2) Continuation semantics subsumes direct semantics in the following sense: For any direct interpretation there is a continuation interpretation (its exact image) and functions $\alpha \in H \to W$ and $\beta \in W \to H$ such that α maps the

direct meaning of any general flow diagram, linear flow diagram, decision table diagram, or process into the continuation meaning, and β maps the continuation meaning back into the direct meaning. Direct semantics does not subsume continuation semantics in a similar sense.

(3) Processes are canonical for continuation, but not direct semantics, since there is a particular continuation interpretation (used to define ΓW^o) which always gives distinct meanings to distinct processes. There is no such direct semantics.

ACKNOWLEDGEMENT

The author would like to thank J. W. Thatcher, J. A. Goguen, and R. Milner for their encouraging and helpful comments on earlier drafts of this paper.

REFERENCES

1. Scott, D., "The Lattice of Flow Diagrams," Symposium on Semantics of Algorithmic Languages, Ed. E. Engeler, Springer Lecture Note Series No. 188, Springer-Verlag, Heidelberg (1971), pp. 311-366. Also, Tech. Monograph PRG-3, Programming Research Group, Oxford University Computing Laboratory, November 1970.
2. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B., "Initial Algebra Semantics and Continuous Algebras," to appear in JACM.
3. Burstall, R. M., and Landin, P. J., "Programs and Their Proofs: an Algebraic Approach", Machine Intelligence 4, ed. B. Meltzer and D. Michie, Edinburgh University Press, 1969, pp. 17-43.
4. Morris, F. L., "The Next 700 Programming Language Descriptions," unpublished.
5. Strachey, C. and Wadsworth, C. P., "Continuations - A Mathematical Semantics for Handling Full Jumps," Tech. Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory, January 1974.
6. Milner, R., "An Approach to the Semantics of Parallel Programs," Proc. Convegno di Informatica Teorica, Pisa, March 1973.

7. Scott, D., "Outline of a Mathematical Theory of Computation," Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems (1970), pp. 169-176. Also, Tech. Monograph PRG-2 Programming Research Group, Oxford University Computing Laboratory, November 1970.

8. _____. "Continuous Lattices," Proc. 1971 Dalhousie Conf., Springer Lecture Note Series, Springer-Verlag, Heidelberg. Also, Tech. Monograph PRG-7, Programming Research Group, Oxford University Computing Laboratory, August 1971.

9. Reynolds, J. C., "Notes on a Lattice-Theoretic Approach to the Theory of Computation", Systems and Information Science, Syracuse University, October 1972.

10. Bekic, H., "Towards a Mathematical Theory of Processes," Technical Report TR25.125, IBM Vienna Laboratory, December 1971.

11. Milner, R., "Implementation and Applications of Scott's Logic for Computable Functions," Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, (or SIGACT News, No. 14), January 1972, pp. 1-6.

12. Egli, H., and Constable, R. L., "Computability Concepts for Programming Language Semantics," Proc. 7th Annual ACM Symposium on Theory of Computing, May 1975, pp. 98-106, also to appear in Theoretical Computer Science.

13. Gordon, M., "Models of Pure LISP," Ph.D. Thesis, Edinburgh University (1973).

14. Plotkin, G. D., "A Powerdomain Construction," to appear in SIAM Journal of Computing.

15. Wand, M., (1974) On the Recursive Specification of Data Types. Category Theory Applied to Computation and Control. Lecture Notes in Computer Science, Vol. 25. Berlin: Springer-Verlag.

16. Scott, D. and Strachey, C., "Towards a Mathematical Semantics for Computer Languages," Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn 21, (1971). Also, Oxford University Computing Laboratory Technical Monograph, PRG 6.

Section 3

USER-DEFINED TYPES AND PROCEDURAL DATA STRUCTURES AS COMPLEMENTARY APPROACHES TO DATA ABSTRACTION

John C. Reynolds

Syracuse University, Syracuse, New York

ABSTRACT User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same kind of data. However, this decentralization of the description of data is achieved at the cost of prohibiting primitive operations from accessing the representations of more than one data item. The contrast between these approaches is illustrated by a simple example.

Work partly supported by NSF Grant GJ-41540.

Introduction

User-defined types and procedural data structures have both been proposed as methods for data abstraction, i.e., for limiting and segregating the portion of a program which depends upon the representation used for some kind of data. In this paper we suggest, by means of a simple example, that these methods are complementary, each providing a capability lacked by the other.

The idea of user-defined types has been developed by Morris,[1,2] Liskov and Zilles,[3] Fischer and Fischer,[4] and Wulf[5], and has its roots in earlier work by Hoare and Dahl.[6] In this approach, each particular conceptual kind of data is called a type, and for each type used in a program, the program is divided into two parts: a type definition and an "outer" or "abstract" program. The type definition specifies the representation to be used for the data type and a set of primitive operations (and perhaps constants), each defined in terms of the representation. The choice of representation is hidden from the outer program by requiring all manipulations of the data type in the outer program to be expressed in terms of the primitive operations. The heart of the matter is that any consistent change in the data representation can be effected by altering the type definition without changing the outer program.

Various notions of procedural (or functional) data structures have been developed by Reynolds,[7] Landin,[8] and Balzer.[9] In this approach, the abstract form of data is characterized by the primitive operations which can be performed upon it, and an item of data is simply a procedure or collection of procedures for performing these operations. The essense of the idea is seen most clearly in its implementation: an item of procedural data is a kind of record called a closure which contains both an internal representation of the data and a pointer (or flag field) to code for procedures for manipulating this representation. A program with access to a closure record is only permitted to examine or access the internal representation by executing the code indicated by the pointer, so that this code serves to close off or protect the internal representation.

In comparison with user-defined types, procedural data structures provide a decentralized form of data abstraction. Each part of the program which creates procedural data will specify its own form of representation, independently of the representations used elsewhere for the same kind of data, and will provide versions of the primitive operations (the components of the procedural data item) suitable

for this representation. There need be no part of the program, corresponding to a type definition, in which all forms of representation for the same kind of data are known. But a price must be paid for this decentralization: a primitive operation can have access to the representation of only a single data item, the item of which the operation is a component.

Apparently this price is inevitable. If an operation is to have access to the representation of more than one item of data, each of which may have several possible representations, then its definition cannot be "decentralized" into one part for each representation, since one must provide for every possible combination of representations. Presumably this requires the definition to occur at a point in the program where all possible representations of the operands are known.

Linguistic Preliminaries

Before illustrating these ideas, we must digress to explain (informally) the language we will use. It is an applicative language, similar to pure LISP [10] or the applicative subsets of GEDANKEN, [7] PAL, [11] or ISWIM, [12] but with a complete type structure somewhat like Algol 68. [13] Types will be indicated by writing $\varepsilon\ \tau$, where τ is a type expression, after binding occurrences of identifiers (except where the type is obvious from context). Type expressions are constructed with the operators $\rightarrow$ denoting functional procedures, $\times$ denoting a Cartesian product, and + denoting a named disjoint union.

The named disjoint union is sufficiently novel to require a more detailed explanation. If $\tau_1, \ldots, \tau_n$ are type expressions denoting the sets $S_1, \ldots, S_n$ and $i_1, \ldots, i_n$ are distinct identifiers, then

$$i_1\colon \tau_1 + \ \ldots\ + i_n\colon \tau_n$$

is a type expression denoting the set of pairs

$$\{\langle i_k, x\rangle \mid 1 \leq k \leq n \text{ and } x \in S_k\}\ .$$

If e is an expression of type τ_k with value x, then

$$\underline{\text{inject}}\ i_k\ e$$

is an expression of type $i_1\colon \tau_1 + \ldots + i_n\colon \tau_n$ with value $\langle i_k, x\rangle$.

Let e be an expression of type $i_1: \tau_1 + \ldots + i_n: \tau_n$ with value <i, x>, let $i_{k_1}, \ldots, i_{k_m}$ be distinct members of the set of identifiers $\{i_1, \ldots, i_n\}$, for $1 \leq j \leq m$ let ℓ_j be an expression of type $\tau_{k_j} \rightarrow \tau'$ with value f_j, and let e' be an expression of type τ' with value x'. Then

$$\underline{\text{unioncase}}\ e\ \underline{\text{of}}\ (i_{k_1}: \ell_1, \ \ldots\ , i_{k_m}: \ell_m, \underline{\text{other}}: e')$$

is an expression of type τ' with the value

$$\underline{\text{if}} \begin{bmatrix} i = i_{k_1} \\ \vdots \\ i = i_{k_m} \\ \text{otherwise} \end{bmatrix} \underline{\text{then}} \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \\ x' \end{bmatrix}$$

When m = n, the other clause will be omitted.

We use the type expression nilset to denote a standard one-element set, whose unique member is denoted by ().

Integer Sets as a User-Defined Type

Our example is an implementation of the abstract concept of sets of integers. Using the approach of user-defined types, we wish to define a type *set* and primitive constants and functions

none ε set

all ε set

limit ε integer × integer × set → set

union ε set × set → set

exists ε integer × integer × set → Boolean

satisfying the specifications

none = {}
all = The set of all (machine-representable) integers
limit(m, n, s) = s ∩ {k | m ≤ k ≤ n}
union(s1, s2) = s1 ∪ s2
when m ≤ n, exists(m, n, s) = (∃ k) m ≤ k ≤ n and k ∈ s

To make our solution seem more realistic, we require that the execution of *limit* and *union* should require time and space bounded by constants which are independent of their arguments. Of course this will exact a price in the speed of *exists*.

An appropriate and simple solution is to represent a set by a list structure which records the way in which the set is constructed via primitive operations. Thus the representation of a set is a disjoint union, over the four set-valued primitive functions (including constants), of sets of possible arguments for these functions. More precisely, this representation is defined by the recursive type declaration:

```
set = nonef: nilset + allf: nilset + limitf: integer × integer × set
        + unionf: set × set
```

and the effect of *none*, *all*, *limit*, or *union* is to imbed its arguments into the appropriate kind of list element:

```
none = inject nonef ()
all = inject allf ()
limit(m, n, s) = inject limitf (m, n, s)
union(s1, s2) = inject unionf (s1, s2)
```

(Roughly speaking, we are representing sets by a free algebra with constants *none* and *all*, and operators *limit* and *union*.) The entire computational burden of interpreting this representation falls upon the function *exists*:

```
exists(m, n, s) = unioncase s of
    (nonef: λ(). false,
    allf: λ(). true,
    limitf: λ(m1, n1, s1). max(m,m1) ≤ min(n,n1)
              and exists(max(m,m1), min(n,n1), s),
    unionf: λ(s1, s2). exists(m, n, s1) or exists(m, n, s2) )
```

(We assume that the operations and and or do not evaluate their second operand when the first operand is sufficient to determine their result.)

Although the above is a definition of the type *set* which meets our specifications, it can be easily improved, even within the time and space constraints imposed upon *limit* and *union*. For example, both *limit* and *union* can be optimized by taking advantage of some obvious properties of sets - the result of *limit* can be simplified when its last argument is *none* or another application of *limit*, and the result of *union* can be simplified when either argument is *none* or *all*:

```
limit(m, n, s) = unioncase s of
     (nonef: λ(). none,
     limitf: λ(m1, n1, s1). if max(m, m1) ≤ min(n, n1)
               then inject limitf (max(m,m1), min(n,n1), s1) else none,
     other: inject limitf (m, n, s)  )
union(s1, s2) = unioncase s1 of
     (nonef: λ(). s2, allf: λ(). all,
     other: unioncase s2 of
          (nonef: λ(). s1, allf: λ(). all,
          other: inject unionf (s1, s2) ))
```

In conclusion, we show how our specification of integer sets might be "packaged" in a language permitting user-defined types:

```
newtype set = nonef: nilset + allf: nilset + limitf: integer × integer × set
    + unionf: set × set
with none ε set = inject nonef (),
    all ε set = inject allf (),
    limit ε integer × integer × set → set =
        λ(m, n, s).unioncase s of
        (nonef: λ(). none,
        limitf: λ(m1, n1, s1). if max (m,m1) ≤ min(n,n1)
            then inject limitf (max(m,m1), min(n,n1), s1) else none,
        other: inject limitf (m, n, s) ),
    union ε set × set → set =
        λ(s1, s2). unioncase s1 of
        (nonef: λ(). s2, allf: λ(). all,
        other: unioncase s2 of
            (nonef: λ(). s1, allf: λ(). all,
            other: inject unionf (s1, s2) )),
    exists ε integer × integer × set → Boolean =
        λ(m, n, s). unioncase s of
        (nonef: λ(). false,
        allf: λ(). true,
        limitf: λ(m1, n1, s1). max(m,m1) ≤ min(n,n1)
            and exists(max(m,m1), min(n,n1), s),
        unionf: λ(s1, s2). exists(m, n, s1) or exists(m, n, s2) )
in <outer program>
```

The language used here is an outgrowth of the ideas discussed in reference 14. A complete exposition of this language is beyond the scope of this paper, but the following salient points should be noted:

(1) The type declaration between newtype and with bin's all occurrences of the type identifier *set* throughout the above expression (including occurrences in <outer program>). The ordinary declarations between with and in bind all occurrences of the ordinary identifiers *none*, *all*, *limit*, *union*, and *exists* throughout the expression.

(2) With regard to occurrences of *set* between with and in, the type declaration behaves like a mode definition in Algol 68, i.e., *set* is equivalent to the type expression on the right side of the type declaration, and the type-correctness of the text in with ... in depends upon this type expression.

(3) In <outer program> occurrences of *set* behave like a primitive type, e.g., integer or Boolean. In other words, <outer program> must be a correctly typed expression regardless of what type expression might be equivalent to *set*. This insures that all manipulations of the user-defined type in <outer program> must be expressed in terms of the primitives declared in with ... in.

(4) Although it is not illustrated by our example, it should be possible to declare simultaneously several related user-defined types between newtype and with. This ability is needed to permit the definition of multiargument primitive functions which act upon more than one user-defined type. An example might be the use of the types *point* and *line* in a program for performing geometrical calculations.

Integer Sets as Procedural Data Structures

We now develop integer sets as procedural data structures. The starting point is the realization that all we ever want to do to a set s, aside from using it to construct other sets, is to evaluate the Boolean expression $exists(m, n, s)$. This suggests that we can simply equate the set s with the Boolean function $\lambda(m, n).\ exists(m, n, s)$ which characterizes the only information we want to extract from the set.

Thus we define

$$\text{set} = \underline{\text{integer}} \times \underline{\text{integer}} \rightarrow \underline{\text{Boolean}}$$

and specify that if $s \in set$ represents the "mathematical" set s_o, then for $m \leq n$,

$$s(m, n) = (\exists\, k)\ m \leq k \leq n \text{ and } k \in s_o.$$

The need for defining the primitive function *exists* has vanished since this function has been internalized - its value for a particular *set* is simply the (only component of the) *set* itself. The remaining primitive constants and functions are easily defined by:

```
none = λ(m, n). false
all = λ(m, n). true
limit(m, n, s) = λ(m1, n1).
    max(m,m1) ≤ min(n,n1) and s(max(m,m1), min(n,n1))
union(s1, s2) = λ(m, n). s1(m, n) or s2(m, n)
```

In this approach, there is no "outer program" from which the definition *set* = integer × integer → Boolean is hidden. Any part of the program can create a *set* by giving an appropriate function whose internal representation (the collection of values of global variables which form the fields of the closure record) can be arbitrary. For example, in augmenting an existing program, one might write

$$\lambda(m, n).\ \text{even}(m)\ \underline{\text{or}}\ (m < n)$$

to denote the set of even integers, or

$$\underline{\text{letrec}}\ s = \lambda(m, n).\ (m \leq n)\ \underline{\text{and}}\ (p(m)\ \underline{\text{or}}\ s(m+1, n))\ \underline{\text{in}}\ s$$

to denote the set of integers satisfying the predicate p. The procedural approach insures that these definitions will mesh correctly with the rest of the program, even though they introduce novel representations.

This kind of extensional capability, which is the main advantage of the procedural approach, is offset by two limitations. In the first place, although (ignoring computability considerations) every set can be represented by a function in integer × integer → Boolean, the converse is false. To represent a set, a function s must satisfy

$$s(m, n) = \bigvee_{k=m}^{n} s(k, k)$$

for all m and n such that $m \leq n$. This kind of condition, which cannot be checked syntactically, must be satisfied by all parts of the program which create sets.

A more important limitation is that only the function *exists*, which has been internalized as (the only component of) a procedural data item, is truly primitive in the sense of having access to the internal representation of a set. Essentially, we have been forced to express the functions *limit* and *union* in terms of the internalized *exists*. We are fortunate that our example permits us to do this at all. Even so, we are prevented from optimizing *limit* and *union* as we did in the user-defined-type development. There is no practically effective

way that *limit*(*m*, *n*, *s*) can "see" whether *s* has the form *none* or *limit*(*m1*, *n1*, *s1*), or that *union*(*s1*, *s2*) can "see" whether *s1* or *s2* has the form *none* or *all*.

In fact, this difficulty can be surmounted for *limit* but not for *union*. The solution is to internalize *limit* as well as *exists*, so that both functions have access to internal representations. Thus we represent sets by pairs of functions:

$$\text{set} = (\underline{\text{integer}} \times \underline{\text{integer}} \to \underline{\text{Boolean}}) \times (\underline{\text{integer}} \times \underline{\text{integer}} \to \text{set})$$

and specify that if *s* represents the mathematical set s_o then for $m \leq n$,

$$s.1(m, n) = (\exists k)\ m \leq k \leq n \text{ and } k \in s_o ,$$

and for all *m* and *n*, *s.2*(*m*, *n*) represents the mathematical set

$$s_o \cap \{k \mid m \leq k \leq n\}$$

(Here *s.1* and *s.2* denote the components of the pair *s*.)

In this approach, we may define *none* by:

```
none = (λ(m, n). false, λ(m, n). none)
```

Note the peculiar kind of recursion which is characteristic of this style of programming: the second component of *none* is a function which does not call itself but rather returns itself as a component of its result.

To define *all* and *union* we first define an "external" *limit* ε integer × integer × *set* → *set* which will be called upon by the internal limiting functions (i.e., the second components) of *all* and *union*:

```
limit(m, n, s) =
     (λ(m1, n1). max(m,m1) ≤ min(n,n1) and s.1(max(m,m1), min(n,n1)),
     λ(m1, n1). if max(m,m1) ≤ min(n,n1) then
               limit(max(m,m1), min(n,n1), s) else none)
```

Then

```
all = (λ(m, n). true, λ(m, n). limit(m, n, all))
union(s1, s2) = (λ(m, n). s1.1(m, n) or s2.1(m, n),
     λ(m, n). limit(m, n, union(s1, s2)) )  .
```

With these definitions, the internal limiting functions perform simplifications analogous to those performed by *limit* in the user-defined-type approach. Indeed, if one examines the behavior of the closures which would represent sets in an implementation of this definition, one finds that they mimic the list structures of the type

approach almost exactly (except for the simplifications performed by union).

But even to someone who is experienced with procedural data structures, the internalization of *limit* is more a tour de force than a specimen of clear programming. Moreover, internalization cannot be applied to give a function such as *union* access to the internal representation of more than one argument, i.e., we could convert *union*(*s1*, *s2*) to a component of *s1* or of *s2* but not both.

Conclusions

In comparison with user-defined types, procedural data structures offer a more decentralized method of data abstraction which precludes any interaction between different representations of the same kind of data. This offers the advantage of easier extensibility at the price of prohibiting primitive operations from accessing the representations of more than one data item.

Of course, the two approaches can be combined. For example, we can augment our user-defined-type definition to include an additional primitive *functset* ε (integer × integer → Boolean) → *set* which accepts a functional set (in the sense of the first part of the previous section) and produces an equivalent value of type *set*. It is sufficient to add one more kind of record to the disjoint union defining *set* and one more alternative to the branches defining *exists*:

```
newtype set = ... + functsetf: (integer × integer → Boolean)
with ...
     functset ε (integer × integer → Boolean) → set = λf. inject functsetf f,
     exists ε integer × integer × set → Boolean =
          λ(m, n, s). unioncase s of
          ( ... functsetf: λf. f(m, n) )
in <outer program>
```

However, this kind of combination is hardly a unification. To some extent, the data-representation structuring approach of Hoare and Dahl[6] unifies the concepts of user-defined types and procedural data structures, but only at the expense of combining their limitations. It appears that this is inevitable; that the two concepts are inherently distinct and complementary.

The reader should be cautioned that this is a working paper describing ongoing research. In particular, the linguistic constructs we have used are tentative and will require considerable study and evolution before they can be integrated into a complete programming language. The extension of these constructs to languages with imperative features is a particularly murky area.

REFERENCES

1. Morris, J. H., Types are not Sets. Proc. ACM Symposium on Principle of Programming Languages, Boston 1973, pp. 120-124.

2. Morris, J. H., Towards More Flexible Type Systems. Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer-Verlag 1974, pp. 377-384.

3. Liskov, B. H. and S. Zilles, "Programming with Abstract Data Types," ACM SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-60. (Also available as MIT Project MAC Computation Structure Group Memo 99).

4. Fischer, A. E., and Fischer, M. J., Mode Modules as Representations of Domains. Proc. ACM Symposium on Principles of Programming Languages, Boston 1973, pp. 139-143.

5. Wulf, W., Alphard: Toward a Language to Support Structured Programs, Department of Computer Science Internal Report, Carnegie-Mellon University, Pittsburgh, Pa., April 1974.

6. Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press 1972.

7. Reynolds, J. C., GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. Comm ACM 13 (May 1970), 308-319.

8. Landin, P. J., A Correspondence Between ALGOL 60 and Church's Lambda-Notation. Comm ACM 8 (February-March 1965), 89-101 and 158-165.

9. Balzer, R. M., Dataless programming.. Proc. AFIPS 1967 Fall Joint Comput. Conf. Vol. 31, MDI Publications, Wayne, Pa., pp. 535-544.

10. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Pt. I. Comm ACM 3, 4 (Apr. 1960), 184-195.

11. Evans, A., PAL - A language designed for teaching programming linguistics. Proc. ACM 23rd Nat. Conf. 1968, Brandin Systems Press, Princeton, N.J., pp. 395-403.

12. Landin, P. J., The next 700 programming languages. Comm ACM 9, 3 (Mar. 1966), 157-166.

13. van Wijngaarden, A. (Ed.), Mailloux, B. J., Pec[illegible], J. E. L., and Koster, C. H. A. Report on the algorithmic language ALGOL 68. MR 101, Mathematisch Centrum, Amsterdam, Feb. 1969.

14. Reynolds, J. C., Towards a Theory of Type Structure. Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer-Verlag 1974, pp. 408-423.

Section 4

TOWARDS A THEORY OF TYPE STRUCTURE

John C. Reynolds

Syracuse University, Syracuse, New York

INTRODUCTION The type structure of programming languages has been the subject of an active development characterized by continued controversy over basic principles.[1-7] In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.[5] We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure. We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular representations used to implement its primitive types. For example, suppose that S and S' are two sets such that the members of S can be used to "represent" the members of S'. We can conceive of running the same program on two machines M and M' in which the same primitive type, say integer, ranges over the sets S and S' respectively. Then if every "integer" input to M represents the corresponding input to M', and if M interprets every primitive operation involving integers in a way which represents the interpretation of M', we expect that every integer output of M should represent the corresponding output of M'. Of course, this idea requires a precise definition of the notion of "represents"; we will supply such a definition after formalizing our illustrative language. The essential thesis of Reference 5 is that this property of representation independence should hold for user-defined types as well as primitive types. The introduction of a user-defined type t should partition a program into

Work partly supported by ARPA (DAHC04-72-C-0003) and NSF (GJ-41540).

an "outer" region in which t behaves like a primitive type and is manipulated by various primitive operations which are used but not defined, and an "inner" region in which the representation of t is defined in terms of other types, and the primitive operations on t are defined in terms of this representation. We expect that the meaning of such a program will remain unchanged if the inner region is altered by changing the representation of the type and redefining its primitive operations in a consistent manner.

We also wish to consider the old but neglected problem of polymorphic functions, originally posed by Strachey. Consider the construction of a program in which several different types of arrays must be sorted. We can conceive of a "polymorphic sort function" which, for any type t, accepts an array with elements of type t and a binary ordering predicate whose arguments must be of type t, and produces an array with elements of type t. We would like to define such a function, and to have each call of the function syntactically checked to insure that it is type-correct for some t. But in a typed language a separate sort function must be defined for each type, while in a typeless language syntactic checking is lost. We suggest that a solution to this problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the syntactic checking of type correctness.

An Illustrative Language

To illustrate these ideas, we introduce an extension of the typed lambda calculus[8] which permits the binding of type variables. Although this language is hardly an adequate vehicle for programming, it seems to pose the essense of the type structure problem, and it is simple enough to permit a brief but rigorous exposition of its semantics.

We begin with a typed lambda calculus in which the type of every expression can be deduced from the type of its free variables. For this purpose it is sufficient to supply, at each point of variable binding, a type expression describing the variable being bound. For example,

$$\lambda x \in t.\ x$$

denotes the identity function for objects of type t, and

$$\lambda f \varepsilon t \to t.\ \lambda x \varepsilon t.\ f(f(x))$$

denotes the doubling functional for functions over t.

It is evident that the meaning of such expressions depends upon both their free normal variables and their free type variables (e.g., t in the above examples). This suggests the addition of a facility for binding type variables to create functions from types to values, called polymorphic functions. For example,

$$\Lambda t.\ \lambda x \varepsilon t.\ x$$

is the polymorphic identity function, which maps t into the identity function for objects of type t, and

$$\Lambda t.\ \lambda f \varepsilon t \to t.\ \lambda x \varepsilon t.\ f(f(x))$$

is the polymorphic doubling functional, which maps t into the doubling functional for functions over t.

The next step is to permit the application of polymorphic functions to type expressions, and to introduce a new form of beta-reduction for such applications. In general, if r is a normal expression and w is a type expression, then

$$(\Lambda t.\ r)[w]$$

denotes the application of the polymorphic function $\Lambda t.\ r$ to the type w, and is reducible to the expression obtained from r by replacing every free occurrence of t by w (after possible alpha-conversion to avoid collision of variables). For example, the application of the polymorphic identity function to the type integer → real,

$$(\Lambda t.\ \lambda x \varepsilon t.\ x)[\underline{\text{integer}} \to \underline{\text{real}}]$$

reduces to the identity functional for functions from integer to real,

$$\lambda x \varepsilon \underline{\text{integer}} \to \underline{\text{real}}.\ x \quad .$$

Finally, we must introduce a new kind of type expre sion to describe the types of polymorphic functions. We write $\Delta t.\ w$ to denote the type of polymorphic function which, when applied to the type t, produces a value of type w. Thus if the expression r has the type w, then the expression $\Lambda t.\ r$ has the type $\Delta t.\ w$. For example, the type of the polymorphic identity function is $\Delta t.\ t \to t$, while the type of the polymorphic doubling functional is $\Delta t.\ (t \to t) \to (t \to t)$.

In providing polymorphic functions, we also provide user-defined types. For example, suppose outer is an expression in which cmp is a primitive type (i.e., a free type variable) intended to denote complex numbers, add and magn are primitive functions (i.e., free normal variables) intended to denote addition and magnitude functions for complex numbers, and i is a primitive constant (i.e., a free normal variable) intended to denote the square root of -1. Suppose we wish to represent complex numbers by pairs of reals, and to represent addition, magnitude, and the square root of -1 by the expressions

addrep ε (real × real) × (real × real) → (real × real)
magnrep ε (real × real) → real
irep ε (real × real)

This representation can be specified by the expression

(λcmp. λadd ε cmp × cmp → cmp. λmagn ε cmp → real. λi ε cmp. outer)
[real × real] (addrep) (magnrep) (irep) .

(Our illustrative language does not include the Cartesian product, but its addition should not pose any significant problems.) Admittedly, this is hard to read, but the problem should be amenable to judicious syntactic sugaring.

We now proceed to develop a formal definition of our illustrative language, culminating in a "representation theorem" which asserts its type correctness.

Notational Preliminaries

For sets S and S', we write S × S' to denote the Cartesian product of S and S', S ⇒ S' or S'^{S} to denote the set of functions from S to S', and when S and S' are domains (in the sense of Scott) S → S' to denote the set of continuous functions from S to S'. If F is a function which maps each member of S into a set, we write $\prod_{x \varepsilon S} F(x)$ to denote the set of functions f such that the domain of f is S and, for each x ε S, f(x) ε F(x).

For f ε S ⇒ S', x ε S, x' ε S', we write [f|x|x'] to denote the function λy ε S. if y = x then x' else f(y).

Syntax

To formalize the syntax of our language, we begin with two disjoint, countably infinite sets: the set T of type variables and the set V of normal variables. Then W, the set of type expressions, is the minimal set satisfying:

(1a) If $t \in T$ then:

$t \in W$.

(1b) If $w_1, w_2 \in W$ then:

$(w_1 \to w_2) \in W$.

(1c) If $t \in T$ and $w \in W$ then:

$(\Delta t.\ w) \in W$.

(To keep the syntax simple, we have specified complete parenthesization, but in writing particular type expressions we will omit parentheses according to common usage.)

From the fact that $\Delta t.\ w$ is supposed to bind the occurrences of t in w, one can define the notions of free and bound occurrences of type variables, and of alpha-conversion of type expressions in an obvious manner. We write $w \simeq w'$ to indicate that w and w' are alpha-convertible. (In a more complex language, the relation $\simeq$ might be larger; the idea is that it must be a decidable equivalence relation which implies that w and w' have the same meaning.)

One can also define the notion of substitution in an obvious manner. We write $w_1 \Big|_t^{w_2}$ to denote the type expression obtained from w_1 by replacing every free occurrence of t by w_2, after alpha-convering w_1 so that no type variable occurs both bound in w_1 and free in w_2.

To define normal expressions, we must capture the idea that every normal expression has an explicit type. Specifically, an assignment of a type expression to every normal variable which occurs free in a normal expression r must induce an assignment of a type expression to r itself which is unique (to within alpha-conversion). For all $Q \in V \Rightarrow W$ and $w \in W$ we write R_{Qw} to denote the set of normal expressions for which the assignment of Q(x) to each normal variable x will induce the assignment of w to the normal expression itself.

Then R_{Qw} is the minimal family of sets satisfying:

(2a) If $Q \in V \Rightarrow W$ and $x \in V$ then:

$$x \in R_{QQ(x)}$$

(2b) If $Q \in V \Rightarrow W$, w_1, w_1', $w_2 \in W$, $w_1 \simeq w_1'$, $r_1 \in R_{Q(w_1 \to w_2)}$, and $r_2 \in R_{Qw_1'}$ then:

$$(r_1\ r_2) \in R_{Qw_2}$$

(2c) If $Q \in V \Rightarrow W$, w_1, $w_2 \in W$, $x \in V$, and $r \in R_{[Q|x|w_1]\ w_2}$ then:

$$(\lambda x \in w_1.\ r) \in R_{Q(w_1 \to w_2)}$$

(2d) If $Q \in V \Rightarrow W$, w_1, $w_2 \in W$, $t \in T$, and $r \in R_{Q(\Delta t.w_1)}$ then:

$$(r[w_2]) \in R_{Q(w_1|_t^{w_2})}$$

(2e) If $Q \in V \Rightarrow W$, $w \in W$, $t \in T$, $r \in R_{Qw}$, and t does not occur free in $Q(x)$ for any x which occurs free in r, then:

$$(\Lambda t.\ r) \in R_{Q(\Delta t.w)}$$

(Again we have specified complete parenthesization, but will omit parentheses according to common usage.) By structural induction on r, it is easy to show that $r \in R_{Qw}$ and $r \in R_{Qw'}$ implies $w \simeq w'$.

The restriction on t in (2e) reflects the fact that the meaning of t in $\Lambda t.\ r$ is distinct from its meaning in the surrounding context. For example, $Q(x) = t$ does not imply $\Lambda t.x \in R_{Q(\Delta t.t)}$.

Semantics

We will interpret our language in terms of the lattice-theoretic approach of D. Scott.[9-12] Intuitively the effect of a type expression is to produce a Scott domain given an assignment of a domain to each free type variable occurring in the type expression. Thus we expect the meaning of type expressions to be given by a function

$$B \in W \Rightarrow \mathcal{D}^T \Rightarrow \mathcal{D}$$

where $\mathcal{D}$ denotes the class of all domains.

To specify B we consider each of the cases in the syntactic definition of W:

(1a) Obviously,

$$B[t](\bar{D}) = \bar{D}(t)$$

(We will use barred variables to denote functions of T, and square brackets to denote application to syntactic arguments.)

(1b) We intend $w_1 \to w_2$ to denote the domain of continuous functions from the domain denoted by w_1 to the domain denoted by w_2. Thus

$$B[w_1 \to w_2](\bar{D}) = \text{arrow}(B[w_1](\bar{D}), B[w_2](\bar{D}))$$

where arrow $\in (\mathcal{D} \times \mathcal{D}) \Rightarrow \mathcal{D}$ satisfies

$$\text{arrow}(D_1, D_2) = D_1 \to D_2.$$

(1c) We intend $\Delta t.\ w$ to denote a set of functions over the class of domains which, when applied to a domain D will produce some element of the domain denoted by w under the assignment of D to t. Thus

$$B[\Delta t.\ w](\bar{D}) = \text{delta}(\lambda D \in \mathcal{D}.\ B[w][\bar{D}|t|D])$$

where delta $\in (\mathcal{D} \Rightarrow \mathcal{D}) \Rightarrow \mathcal{D}$ satisfies

$$\text{delta}(\theta) \subseteq \prod_{D \in \mathcal{D}} \theta(D)$$

We leave open the possibility that delta(θ) may be a proper subset of the above expression. (Indeed, if we are going to avoid the paradoxes of set theory and consider delta(θ) to be a domain, it had better be a very proper subset.)

By structural induction, one can show that $w \simeq w'$ implies $B[w] = B[w']$, and that

$$B[w_1 \,|_t^{w_2}](\bar{D}) = B[w_1][\bar{D} \mid t \mid B[w_2](\bar{D})]$$

The effect of a normal expression is to produce a value, given an assignment of domains to its free type variables and an assignment of values to its free normal variables. (We will call the latter assignment an environment.) However, this effect must conform to the type structure. When given a type assignment $\bar{D}$, a normal expression $r \in R_{Qw}$ must only accept environments which map each variable x into a member of the domain $B[Q(x)](\bar{D})$, and r must produce a member of the domain $B[w](\bar{D})$. Thus we expect that, for all $Q \in V \Rightarrow W$ and $w \in W$, the meaning of the normal expressions in R_{Qw} will be given by a function

$$M_{Qw} \in R_{Qw} \Rightarrow \prod_{\bar{D} \in \mathcal{D}^T} (Env_Q(\bar{D}) \rightarrow B[w](\bar{D}))$$

where

$$Env_Q(\bar{D}) = \prod_{x \in V} B[Q(x)](\bar{D}) .$$

To specify the M_{Qw} we consider each of the cases in the syntactic definition of R_{Qw}. Essentially the specification is an immediate consequence of the intuitive meaning of the language, guided by the necessity of making the functionalities come out right:

(2a) $M_{QQ(x)}[x](\bar{D})(e) = e(x)$

(2b) $M_{Qw_2}[r_1\ r_2](\bar{D})(e) = (M_{Q(w_1 \rightarrow w_2)}[r_1](\bar{D})(e))(M_{Qw_1}[r_2](\bar{D})(e))$

(2c) $M_{Q(w_1 \rightarrow w_2)}[\lambda x \in w_1.\ r](\bar{D})(e)$

$= \lambda a \in B[w_1](\bar{D}).\ M_{[Q|x|w_1]w_2}[r](\bar{D})[e|x|a]$

(2d) $M_{Q(w_1|_t^{w_2})}[r[w_2]](\bar{D})(e) = (M_{Q(\Delta t.w_1)}[r](\bar{D})(e))(B[w_2](\bar{D}))$

(2e) $M_{Q(\Delta t.w)}[\Lambda t.\ r](\bar{D})(e) = \lambda D \in \mathcal{D}.\ M_{Qw}[r][\bar{D}|t|D](e)$

Representations

Before we can formulate the representation theorem, we must specify what we mean by representation.

For $D, D' \in \mathbb{D}$, the set of representations between D and D', written $\mathrm{rep}(D, D')$, is the set of continuous function pairs

$$\mathrm{rep}(D, D') = \{ \langle \phi, \psi \rangle \mid \phi \in D \to D', \psi \in D' \to D, \psi \cdot \phi \sqsupseteq I_D, \phi \cdot \psi \sqsubseteq I_{D'} \}$$

where I_D denotes the identity function on D. For $x \in D$, $x' \in D'$, and $\rho = \langle \phi, \psi \rangle \in \mathrm{rep}(D, D')$, we write

$$\rho: x \mapsto x'$$

and say that x represents x' according to ρ if and only if

$$x \sqsubseteq \psi(x')$$

or equivalently,

$$\phi(x) \sqsubseteq x' \quad .$$

A pragmatic justification of this rather ad hoc definition is that it will ultimately make the representation theorem correct. (Although this would still be true if we took $\mathrm{rep}(D, D')$ to be the set of projection pairs between D and D', i.e., if we replaced the requirement $\psi \cdot \phi \sqsupseteq I_D$ by $\psi \cdot \phi = I_D$.) However, some intuition is provided by the following connection with the notion of representation between sets. Conventionally, we might say that a representation between a set S and a set S' is simply a function $\pi \in S \rightleftharpoons S'$, and that $x \in S$ represents $x' \in S'$ according to π iff $\pi(x) = x'$. But if we take D and D' to be the powerset domains 2^S and $2^{S'}$ (with $\subseteq$ as $\sqsubseteq$), and ϕ and ψ to be the pointwise extensions of π and its converse (as a relation), then $\rho = \langle \phi, \psi \rangle$ is a representation between D and D', and $\rho: s \mapsto s'$ iff every $x \in s$ conventionally represents some $x' \in s'$ according to π.

The following is an obvious and useful extension of our definition. For $\overline{D}, \overline{D}' \in \mathbb{D}^T$, we define

$$\mathrm{rep}(\overline{D}, \overline{D}') = \prod_{t \in T} \mathrm{rep}(\overline{D}(t), \overline{D}'(t)) \quad .$$

The Representation Theorem

At this point, we can formulate a preliminary version of the representation theorem. Consider the set of normal expressions R_{Qw}, and suppose that $\overline{D}$, $\overline{D}' \in \mathcal{D}^T$ and $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, so that for each type variable t, $\overline{\rho}(t)$ is a representation between the domains $\overline{D}(t)$ and $\overline{D}'(t)$. Moreover, suppose that e and e' are environments such that, for each normal variable x, e(x) represents e'(x) according to the relevant representation, i.e., $\overline{\rho}(Q(x))$. Then we expect that the value of any $r \in R_{Qw}$ when evaluated with respect to $\overline{D}$ and e should represent the value of the same normal expression when evaluated with respect to $\overline{D}'$ and e', according to the relevant representation, i.e., $\overline{\rho}(w)$.
More formally:

Let $Q \in V \Rightarrow W$, $w \in W$, $\overline{D}$, $\overline{D}' \in \mathcal{D}^T$, $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, $e \in Env_Q(\overline{D})$, and $e' \in Env_Q(\overline{D}')$. If

$$(\forall x \in V)\ \overline{\rho}(Q(x)): e(x) \mapsto e'(x)$$

then

$$(\forall r \in R_{Qw})\ \ \overline{\rho}(w): M_{Qw}[r](\overline{D})(e) \mapsto M_{Qw}[r](\overline{D}')(e')$$

However, this formulation has a serious flaw. In choosing $\overline{\rho}$, we assign a representation to every type variable, but not to every type expression, so that the representations $\overline{\rho}(Q(x))$ and $\overline{\rho}(w)$ are not fully defined. Moreover, we can hardly expect to assign an arbitary representation to every type expression. For example, once we have chosen a representation for integer and a representation for real, we would expect that this choice would determine a representation for integer → real and for any other type expression constructed from integer and real.

In brief, we have underestimated the meaning of type expressions. Not only must B[w] map an assignment of domains to type variables into a domain, but it must also map an assignment of representations into a representation. If we can extend the meaning of B to do so, then a correct formulation of the representation theorem is:

Let $Q \in V \Rightarrow W$, $w \in W$, $\overline{D}$, $\overline{D}' \in \mathcal{D}^T$, $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, $e \in Env_Q(\overline{D})$, and $e' \in Env_Q(\overline{D}')$. If

$$(\forall x \in V)\ B[Q(x)](\overline{\rho}): e(x) \mapsto e'(x)$$

then

$$(\forall r \in R_{Qw})\ B[w](\overline{\rho}): M_{Qw}[r](\overline{D})(e) \mapsto M_{Qw}[r](\overline{D}')(e')$$

The Full Semantics of Type Expressions

In order to extend the semantic function B, we first note that the combination of domains and representations forms a category. We write C to denote the category, called the category of types, in which the set of objects is $\mathfrak{D}$, the set of morphisms from D to D' is rep(D, D'), composition is given by

$$\langle \phi', \psi' \rangle \cdot \langle \phi, \psi \rangle = \langle \phi' \cdot \phi, \psi \cdot \psi' \rangle$$

and the identity for D is

$$\mathfrak{d}_D = \langle I_D, I_D \rangle \quad .$$

From the category of types, we can form two further categories by standard constructions of category theory.[13] We write C^T to denote the category in which the set of objects is $\mathfrak{D}^T$, the set of morphisms from $\overline{D}$ to $\overline{D}'$ is rep($\overline{D}$, $\overline{D}'$), composition is given by

$$(\overline{\rho}' \cdot \overline{\rho})(t) = \overline{\rho}'(t) \cdot \overline{\rho}(t)$$

and the identity for $\overline{D}$ is given by

$$\mathfrak{d}_{\overline{D}}(t) = \mathfrak{d}_{\overline{D}(t)} \quad .$$

We write Funct(C, C) to denote the category in which the objects are the functors from C to C and the morphisms from θ to θ' are the natural transformations from θ to θ', i.e., the functions $\eta \in \prod_{D \in \mathfrak{D}} \text{rep}(\theta(D), \theta'(D))$ such that, for all D, D' $\in \mathfrak{D}$ and $\rho \in$ rep(D, D'),

$$\theta'(\rho) \cdot \eta(D) = \eta(D') \cdot \theta(\rho)$$

Composition is given by

$$(\eta' \cdot \eta)(D) = \eta'(D) \cdot \eta(D)$$

and the identity for θ is given by

$$\mathfrak{d}_\theta(D) = \mathfrak{d}_{\theta(D)} \quad .$$

We have seen that the meaning B[w] of a type expression w must map the objects of C^T into the objects of C and the morphisms of C^T into the morphisms of C. Moreover, if our formulation of the representation theorem is to be meaningful, we must have

$$\bar{\rho} \in \text{rep}(\bar{D}, \bar{D}') \text{ implies } B[w](\bar{\rho}) \in \text{rep}(B[w](\bar{D}), B[w](\bar{D}'))$$

Thus, at least if we can satisfy the appropriate laws, we expect to extend B[w] from a function from $\mathcal{D}^T$ to $\mathcal{D}$ into a functor from C^T to C.

Indeed, by pursuing the analogy of categories with sets and functors with functions, we can induce the main structure of the definition of B[w]. For each of the cases in the syntactic definition of W:

(1a) $B[t](\bar{D}) = \bar{D}(t)$

$B[t](\bar{\rho}) = \bar{\rho}(t)$

(1b) $B[w_1 \to w_2](\bar{D}) = \text{arrow}(B[w_1](\bar{D}), B[w_2](\bar{D}))$

$B[w_1 \to w_2](\bar{\rho}) = \text{arrow}(B[w_1](\bar{\rho}), B[w_2](\bar{\rho}))$

where arrow is a bifunctor from $C \times C$ into C.

(1c) $B[\Delta t.\ w] = \text{delta} \cdot \text{abstract}$

where abstract is the functor from C^T into Funct(C, C) such that

$\text{abstract}(\bar{D})(D) = B[w][\bar{D}|t|D]$

$\text{abstract}(\bar{D})(\rho) = B[w][\phi_{\bar{D}}|t|\rho]$

$\text{abstract}(\bar{\rho})(D) = B[w][\bar{\rho}|t|\phi_D]$

and delta is a functor from Funct(C, C) into C.

Even before defining the functors arrow and delta, it can be shown that B maps every type expression into a functor from C^T into C, that $w \simeq w'$ implies $B[w] = B[w']$, and that

$$B[w_1|_t^{w_2}](\bar{D}) = B[w_1][\ \bar{D} \mid t \mid B[w_2](\bar{D})\ ]$$

$$B[w_1|_t^{w_2}](\bar{\rho}) = B[w_1][\ \bar{\rho} \mid t \mid B[w_2](\bar{\rho})\ ]$$

The Functors arrow and delta

The definition of the functor arrow is fairly obvious. Essentially, its action on representations is to produce the only reasonable composition which matches domains correctly: For all $D_1, D_2 \in \mathcal{D}$,

$$\text{arrow}(D_1, D_2) = D_1 \to D_2 \quad .$$

For all $\langle \phi_1, \psi_1 \rangle \in \text{rep}(D_1, D_1')$ and $\langle \phi_2, \psi_2 \rangle \in \text{rep}(D_2, D_2')$,

$$\text{arrow}(\langle \phi_1, \psi_1 \rangle, \langle \phi_2, \psi_2 \rangle) =$$

$$\langle \lambda f \in D_1 \to D_2 .\ \phi_2 \cdot f \cdot \psi_1,\ \lambda f \in D_1' \to D_2' .\ \psi_2 \cdot f \cdot \phi_1 \rangle$$

(The action of arrow on representations is similar to the method used by Scott to construct retraction or projection pairs for function spaces.)

The definition of arrow and the properties of representations give the following lemma:

Let $f \in D_1 \to D_2$, $f' \in D_1' \to D_2'$, $\rho_1 \in \text{rep}(D_1, D_1')$, and $\rho_2 \in \text{rep}(D_2, D_2')$. Then

$$\text{arrow}(\rho_1, \rho_2): f \mapsto f'$$

if and only if, for all $x \in D_1$ and $x' \in D_1'$,

$$\rho_1: x \mapsto x' \text{ implies } \rho_2: f(x) \mapsto f'(x') \ .$$

which, with the definition of B, gives the following lemma:

Let $w_1, w_2 \in W$, $\bar{\rho} \in \text{rep}(\bar{D}, \bar{D}')$, $f \in B[w_1 \to w_2](\bar{D})$, and $f' \in B[w_1 \to w_2](\bar{D}')$. Then

$$B[w_1 \to w_2](\bar{\rho}): f \mapsto f'$$

if and only if, for all $x \in B[w_1](\bar{D})$ and $x' \in B[w_1](\bar{D}')$,

$$B[w_1](\bar{\rho}): x \mapsto x' \text{ implies } B[w_2](\bar{\rho}): f(x) \mapsto f'(x') \quad .$$

(As an aside, we note that the definition of arrow establishes a connection between our notion of representation and the concept of simulation.[14] Typically, one says that a function $\pi \in S \rightrightarrows S'$ is a simulation of a relation $r \subseteq S \times S$ by a relation $r' \subseteq S' \times S'$ iff $\pi \cdot r \subseteq r' \cdot \pi$ (where $\cdot$ denotes relational composition). But if f, f', ϕ, and ψ are the pointwise extensions of r, r', π, and the converse of π, then $\pi \cdot r \subseteq r' \cdot \pi$ iff $\text{arrow}(\rho, \rho): f \mapsto f'$, where $\rho = \langle \phi, \psi \rangle$.)

The definition of the functor delta is less obvious. For all functors θ from C to C, delta(θ) is the complete lattice with elements

$$\{ f \mid f \in \prod_{D \in \mathcal{D}} \theta(D) \text{ and } (\forall D, D' \in \mathcal{D})(\forall \rho \in rep(D,D'))\theta(\rho): f(D) \mapsto f(D') \}$$

with the partial ordering $f \sqsubseteq g$ iff $(\forall D \in \mathcal{D})\ f(D) \sqsubseteq_{\theta(D)} g(D)$. For all natural transformations η from θ to θ',

$$delta(\eta) = \langle \lambda f \in delta(\theta).\ \lambda D \in \mathcal{D}.\ [\eta(D)]_1(f(D)),\ \lambda f \in delta(\theta').\ \lambda D \in \mathcal{D}.\ [\eta(D)]_2(f(D)) \rangle .$$

At this point, we must admit a serious lacuna in our chain of argument. Although delta(θ) is a complete lattice (with $(\bigsqcup F)(D) = \bigsqcup_{\theta(D)} \{f(D) \mid f \in F\}$), it is not known to be a domain, i.e., the question of whether it is continuous and countably based has not been resolved. Nevertheless there is reasonable hope of evading the set-theoretic paradoxes. Even though $\prod_{D \in \mathcal{D}} \theta(D)$ is immense (since $\mathcal{D}$ is a class), the stringent restrictions on membership in delta(θ) seem to make its size tractable. For example, if $f \in delta(\theta)$, then the value of f(D) determines its value for any domain isomorphic to D.

The definition of delta and the properties of representations give the lemma:

Let η be a natural transformation from θ to θ', $f \in delta(\theta)$ and $f' \in delta(\theta')$. Then

$$delta(\eta): f \mapsto f'$$

if and only if, for all D, $D' \in \mathcal{D}$, and $\rho \in rep(D, D')$,

$$\eta(D') \cdot \theta(\rho): f(D) \mapsto f'(D') .$$

which, with the definition of B, gives:

Let $t \in T$, $w \in W$, $\bar{\rho} \in rep(\bar{D}, \bar{D}')$, $f \in B[\Delta t.\ w](\bar{D})$, and $f' \in B[\Delta t.\ w](\bar{D}')$. Then

$$B[\Delta t.\ w](\bar{\rho}): f \mapsto f'$$

if and only if, for all D, $D' \in \mathcal{D}$, and $\rho \in rep(D, D')$,

$$B[w][\bar{\rho} \mid t \mid \rho]: f(D) \mapsto f'(D') .$$

From the final lemmas obtained about arrow and delta, the representation theorem can be proved by structural induction on ι.

Some Syntactic Manipulations

We have explored our illustrative language semantically rather than syntactically, i.e., we have provided it with a mathematical meaning instead of investigating the syntactic consequences of reducibility. However, an obvious question is raised by the fact that every expression in the typed lambda calculus, but not the untyped lambda calculus, has a normal form.[8]

We have been unable to resolve this question for our language. Nevertheless, the language permits some interesting constructions which are not possible in the typed lambda calculus.

For example, consider the following normal expressions:

$$\rho_n \equiv \Lambda t.\ \lambda f \in t \to t.\ \lambda x \in t.\ \underbrace{f(\ \ldots\ f}_{n \text{ times}}(x)\ \ldots\ )$$

of type $\pi \equiv \Delta t.\ (t \to t) \to (t \to t)$,

$$\theta \equiv \lambda h \in \pi.\ \Lambda t.\ \lambda f \in t \to t.\ \lambda x \in t.\ f(h[t]\ f\ x)$$

of type $\pi \to \pi$ (We assume application is left-associative.),

$$\alpha \equiv \lambda g \in \pi \to \pi.\ \lambda h \in \pi.\ g(h[\pi]\ g\ \rho_1)$$

of type $(\pi \to \pi) \to (\pi \to \pi)$, and

$$\beta \equiv \lambda w \in \pi.\ w[\pi \to \pi]\ \alpha\ \theta$$

of type $\pi \to (\pi \to \pi)$. Then the following expressions are interconvertible:

$$\theta\ \rho_n \simeq \rho_{n+1}$$
$$\beta\ \rho_{m+1} \simeq \alpha\ (\beta\ \rho_m)$$
$$\beta\ \rho_0\ \rho_n \simeq \rho_{n+1}$$
$$\beta\ \rho_{m+1}\ \rho_0 \simeq \beta\ \rho_m\ \rho_1$$
$$\beta\ \rho_{m+1}\ \rho_{n+1} \simeq \beta\ \rho_m\ (\beta\ \rho_{m+1}\ \rho_n) \quad .$$

From the last three equations it follows that $\beta\ \rho_m\ \rho_n \sim \rho_{\psi(n,m)}$, where $\psi(n,m)$ is Ackermann's function.

Further Remarks

Since the writing of the preliminary version of this paper, considerable attention has been given to the "serious lacuna" mentioned above. We have managed to show that delta(θ) is a continuous lattice, but not that it is countably based. Conceivably, our notion of representation is too restrictive, which would tend to make delta(θ) unnecessarily large.

ACKNOWLEDGEMENT

The author would like to thank Dr. Lockwood Morris for numerous helpful suggestions and considerable encouragement.

REFERENCES

1. Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A., Report on the Algorithmic Language ALGOL 68. MR 101 Mathematisch Centrum, Amsterdam, October 1969. Also Numerische Mathematik 14 (1969) 79-218.

2. Cheatham, T. E., Jr., Fischer, A., and Jorrand, P., On the Basis for ELF-An Extensible Language Facility. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33 Pt. 2, MDI Publications, Wayne, Pa., pp. 937-948.

3. Reynolds, J. C., A Set-theoretic Approach to the Concept of Type. Working paper, NATO Conf. on Techniques in Software Engineering, Rome, October 1969.

4. Morris, J. H., "Protection in Programming Languages," Comm. ACM, 16 (1), January 1973.

5. Morris, J. H., Types are not Sets. Proc. ACM Symposium on Principle of Programming Languages, Boston 1973, pp. 120-124.

6. Fischer, A. E., and Fischer, M. J., Mode Modules as Representations of Domains. Proc. ACM Symposium on Principles of Programming Languages, Boston 1973, pp. 139-143.

7. Liskov, B., and Zilles, S., An Approach to Abstraction. Computation Structures Group Memo 88, Project MAC, MIT, September 1973.

8. Morris, J. H., Lambda-calculus Models of Programming Languages. MAC-TR-57, Project MAC, MIT, Cambridge, Mass., December 1968.

9. Scott, D., "Outline of a Mathematical Theory of Computation," Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems (1970), pp. 169-176. Also, Tech. Monograph PRG-2, Programming Research Group, Oxford University Computing Laboratory, November 1970.

10. _____. "Continuous Lattices," Proc. 1971 Dalhousie Conf., Springer Lecture Note Series, Springer-Verlag, Heidelberg. Also, Tech. Monograph PRG-7, Programming Research Group, Oxford University Computing Laboratory, August 1971.

11. _____. "Mathematical Concepts in Programming Language Semantics," AFIPS Conference Proc., Vol. 40, AFIPS Press, Montvale, New Jersey (1972), pp. 225-234.

12. _____. "Data Types as Lattices", Notes, Amsterdam, June 1972.

13. MacLane, S., Categories for the Working Mathematician, Springer-Verlag, New York 1971.

14. Morris, F. L., Correctness of Translations of Programming Languages -- An Algebraic Approach, Stanford Computer Science Department Report STAN-CS-72-303, August 1972.

Section 5

An Introduction to Transaction Processing Systems

Daniel K. Wood and Robert G. Sargent

ABSTRACT

An introduction to Transaction Processing Systems is given. Their characteristics, hardware and software requirements, a design methodology, their current development and areas needing future research are briefly described.

TABLE OF CONTENTS

I. INTRODUCTION

The use of computers in processing numeric and non-numeric data has advanced significantly since their introduction. This has resulted in most organizations becoming dependent upon their use. Computerized systems have been developed for all kinds of applications, have increased in sophistication, and have reduced the manhours required to perform numerous kinds of operations. Several different kinds of computer systems have been developed such as batch, on-line, time-sharing, and minis. With the rapid technology changes taking place in computer systems today, one cannot anticipate the kind of computer systems that will result in the future nor the various uses they will be put to.

Computerized systems have had a major effect on how many organizations operate. These systems usually process and handle operational data and information entirely different than was used previously and provide information that was previously unattainable. One type of computerized system causing operating changes are Transaction Processing Systems (TPS). These systems are rapidly being developed and implemented in a wide variety of applications. Most TPS are on-line interactive systems that process business transactions immediately. Examples of such systems are airline and hotel reservation systems, bank credit systems, and certain inventory-ordering systems. This paper provides an introduction to transaction processing systems and discusses some areas needing research.

AD-A054 942 SYRACUSE UNIV N Y F/G 9/2
LARGE SCALE INFORMATION SYSTEMS. VOLUME I.(U)
MAR 78 F30602-74-C-0335
UNCLASSIFIED RADC-TR-78-43-VOL-1 NL

2 OF 2
AD
A054942

END
DATE
FILMED
7-78
DDC

II. CHARACTERISTICS OF TRANSACTION PROCESSING SYSTEMS

Transaction Processing Systems (TPS) generally have certain common characteristics. They are the following [2]:

(a) The input data, called transactions, are of pre-defined types.

(b) The data entry is through terminals and the operating procedure is usually quite simple to allow "non-experts" to use such systems.

(c) The programs to process the transactions are prepared and stored in the computer in advance and are "invisible" to the user.

(d) The response time is fast providing the user with the desired information and/or verifying the input immediately.

(e) The system maintains an integrated data base. The processing of transactions usually involves updating, retrieving, manipulating, or entering new data in the data base.

Simply stated, most TPS are a specialized type of on-line system that allow non-experts to use them interactively for processing pre-defined transactions using previously written stored programs to interact with a data base.

III. PROCESSING OF TRANSACTIONS

After transactions (data) are generated in the organization's operation, they are entered into the computer through a terminal that is

either hardwired or connected by a telephone line. A front-end processor is usually linked between the terminal and the computer. This processor typically screens the transaction for legitimacy, forwards them to the computer, coordinates the arrival and departure of transactions, and may provide editing capability of incoming data. After the transactions pass through the front end processor to the computer, they are assigned priorities based upon their transaction type and urgency of the processed results. These transactions are then queued for service by the CPU. When the CPU becomes available, the computer makes available the necessary application programs to process the first transaction and then executes these programs.

The executing of transaction processing application programs generally involve some interaction with the computer managed data base. The simplest interaction is just to retrieve some information stored in the data base. Complicated interactions may include several updates (insertion, alternation or deletion) of the records in different files. Numerical calculation may be an integral part of the processing of transaction data. Usually, the logic is straightforward, even though it may not be trivial. The time required for processing different types of transactions will be the total time needed in loading the application programs, accessing and storing data from the data base files, and numerical/non-numerical manipulation of data. This will generally take much less time than one second. Since this time is relatively small, the technique of swapping programs in and out of main storage during

execution is rarely used. Once the transaction is entered into the system and validated, all processing of this transaction is carried through to completion, and the output is transmitted back to the terminal.

During the execution of the application programs, if any exceptional conditions occur, the proper procedures for handling these conditions are automatically called for and executed. This may result in reports being generated and special messages being sent. For example, during the processing of an order request, the amount of inventory may fall below a specific level causing a request for an order to be printed out by calling for and executing a special program.

IV. TRANSACTION PROCESSING CONTROL SYSTEM

A software system is needed to perform and control the tasks necessary for a TPS. These software systems (programs) are called such names as transaction (processing) control system, transaction processing (operating) executive, teleprocessing monitor, and transaction (processing) monitor. The tasks of concern are generally the following [8]:

--Terminal management - Host and effectively control the data transfer between a telecommunication network of heterogeneous mix of terminals and the application programs.

--Multitask control - Schedule and support the concurrent processing of a number of application programs for the wide mixture of transaction workloads.

--File management - Provide efficient access methods in handling

data base files. Support the scheduling and initiation of all file item requests made by the application programs.

--Storage management - Allocate and control of storage for programs, storage for input/output buffering areas and temporary storage of data.

--Program management - Provide a multiprogramming capability while offering a real-time program fetch capability. Intercept program interrupts to prevent total system termination.

--Error handling - Detect and handle error conditions caused by hardware or software. Provide dump facility to assist in analysis of programs and transaction undergoing development or modification.

In satisfying these tasks the following services are generally provided for:

1. Edit the input transaction data.
2. Check transaction input for validity.
3. Journalize transaction inputs for use as backup trail.
4. Analyze transaction inputs and establish the linkage of data to proper application programs.
5. Schedule and control the processing of application programs as required by the transaction type.
6. Control of the data transfer between application programs and terminals.
7. Handle exception conditions.

8. Manage the data transfer to and from the data base files as required by the application programs.

9. Manage the application programs in the memory store.

10. Generate reports as required by the application.

11. Generate intermediate transaction data for later processing.

12. Manage the transient data generated during execution of application programs.

13. Monitor the creation of application programs.

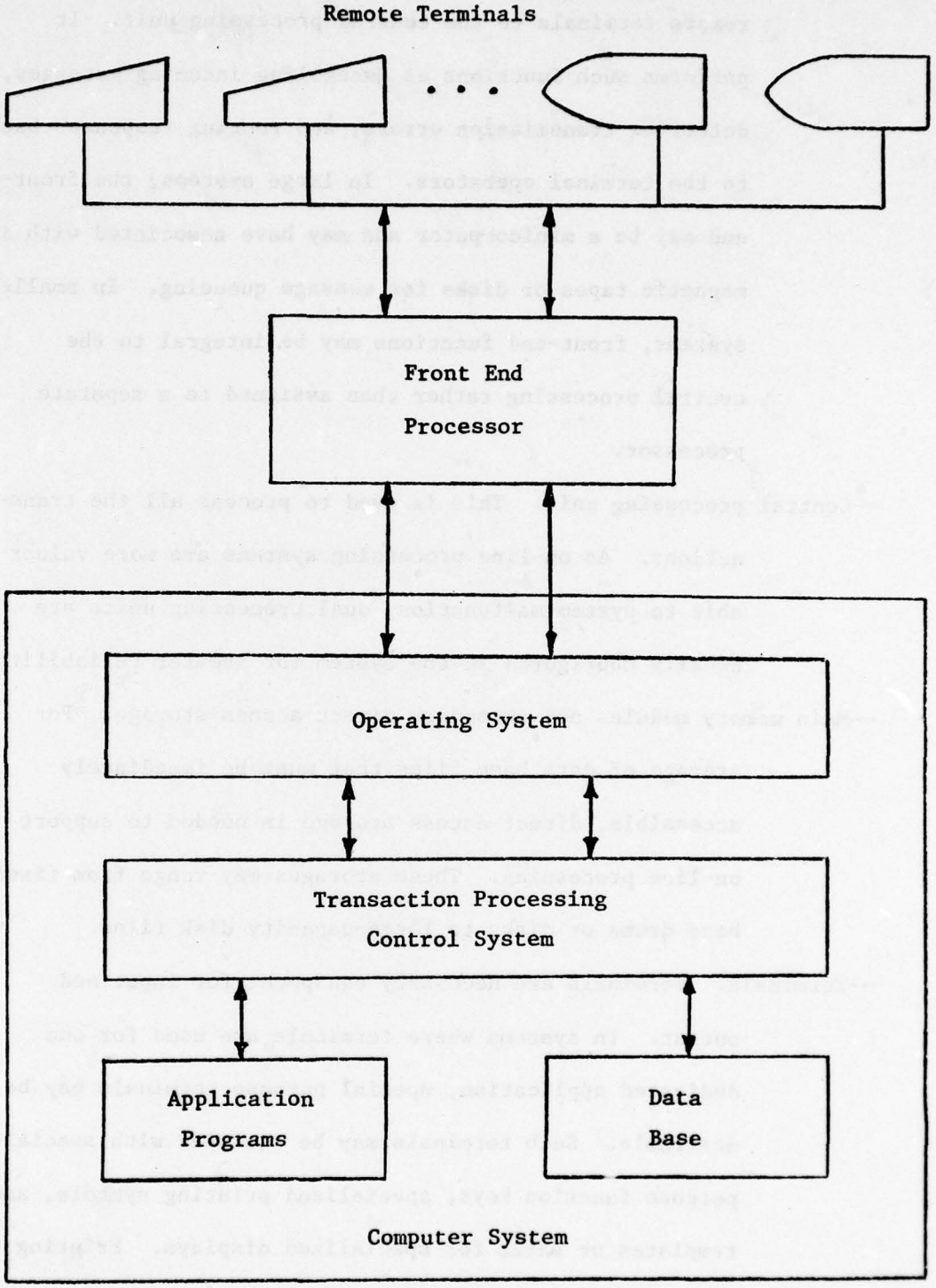
These software systems usually run under the operating system in a multiprogrammed environment as a single privileged job. There are, however, applications where the whole computer system is designed and operated as a dedicated TPS. In these applications, the transaction processing control system is merged into the operating system of the computer system.

V. HARDWARE AND SOFTWARE REQUIREMENTS

Figure 1 is a pictorial representation of a general purpose transaction processing system. To have such a TPS, certain hardware and software are necessary. Since each TPS can be separately designed, there may be some differences between the software and hardware of each.

The following equipment (hardware) are generally requi ed:

--Front end processor. A front end processor (communication front end) is necessary to interface the lines connecting



A General Purpose Transaction Processing System

FIGURE 1

remote terminals to the central processing unit. It performs such functions as assembling incoming messages, detecting transmission errors, and routing responses back to the terminal operators. In large systems, the front-end may be a minicomputer and may have associated with it magnetic tapes or disks for message queueing. In smaller systems, front-end functions may be integral to the central processing rather than assigned to a separate processor.

--Central processing unit. This is used to process all the transactions. As on-line processing systems are more vulnerable to system malfunction, dual processing units are usually configured in the system for greater reliability.

--Main memory modules and secondary direct access storage. For storage of data base files that must be immediately accessible, direct access storage is needed to support on-line processing. These storages may range from fixed-head drums or disks to large-capacity disk files.

--Terminals. Terminals are necessary equipment for input and output. In systems where terminals are used for one dedicated application, special purpose terminals may be desirable. Such terminals may be equipped with special-purpose function keys, specialized printing symbols, and templates or masks for specialized displays. Printing

outputs on predesigned forms can be done directly through the terminal as a result of processing the transaction application programs. These terminals further reduce the works of the terminal operators and eliminate possibilities of typing errors, hence the rate of processing transaction may be increased. For systems that support many diversified applications and handling of relatively unrestricted retrieval requests are expected, general purpose terminals may be more desirable. A standard low-speed Teletype device or an input keyboard combined with a data display screen will suffice for input transaction data as well as receiving processed output. When the applications are complex, minicomputers and microprocessors are quite often used as "intelligent terminals." They are equipped with enough storage capacity and logic so that extensive preliminary processing can be performed before transmitting a transaction to the central processing unit.

The software support needed includes:

--The operating system. Operating system is needed to control other software packages designed for various services. It maintains overall control of system operations by scheduling the execution of all other programs, allocating main memory, establishing job priorities, servicing interrupts, communicating with the computer operator, and performing similar housekeeping tasks.

--Transaction Processing Control System. This software system performs and controls the tasks necessary to operate the TPS as described above. It may include the front end control system and the data base management system listed below.

--Front end control system. This software system is required to control the operation of the multiple remote terminals. Besides controlling the terminals and communication lines, and interrupting the operating system when action must be taken on incoming messages, it may also perform such functions as input message error handling, input editing, output report and display formatting. Features such as provisions of security and data logging capability as well as other supports to facilitate recovery may also be obtained in this software system.

--Data base management system. The data base management system services the application programs and the system users by providing retrieval and updating capability for the file records of an on-line system. It is designed to efficiently handle file and data access methods for manipulating the records in data base. Other functional supports may include maintaining the integrity of data and protect the data base against unauthorized access or modification.

VI. DESIGNING TRANSACTION PROCESSING SYSTEMS

Prior to designing and developing an actual transaction processing system, the following analysis should be conducted to provide information about the characteristics of the transactions, the data base files, and the processing functions.

1. Review the nature of transactions, including frequency and information content of each type of transaction.
2. Establish the required content of output.
3. Define output requirements.
4. Identify the input information needed to produce the required output.
5. Determine the source of data input needed to produce the output.
6. Determine the method of data entry.
7. Determine the data base size, information content and accessing activities.
8. Determine the conditions when information should be processed.
9. Determine the processing algorithms.

Given a specific application, the following is an outline of stages in the process of designing transaction processing systems [3]. This process is iterative and a decision at any stage can have a significant effect on the final design. On the other hand, it is also affected by the previous taken decisions.

1. Select logical record contents - select the items making up the record and determine the key for identifying the record.
2. Rationalize logical records - examine logical records to determine the possiblity of combining or altering record contents in order to simplify the data structure.
3. Select record format - determine the length of records and decide whether the record should be of fixed or variable length.
4. Select file organization - choose the file organization methods among indexed sequential, multi-list randomized addressing or directing.
5. Consideration of logical protection of master files to assure the continued accuracy and completeness of data files as well as to guarantee that the information held in the files will not be disclosed to unauthorized users.
6. Processing mode of master file consideration - determine whether random processing (processing in the order of incoming transactions) or reorder the transactions according to the records in logical sequence is more advantageous.
7. Dialogue design - select among the following alternatives in man-machine dialogue:
   a) dialogue using mnemonics.
   b) computer-initiated dialogue ("menu-selecting").
   c) form-filling.

d) mixture of computer-initiated and operator-initiated dialogue ("hybrid dialogue").

8. Communication network design - examine the peak volume of transactions to be handled in different locations and determine the number of terminals needed in each location.
9. Establish hardware and software environment - establish the mechanism of task operations (transaction processing control system)--i.e., its method of dispatching tasks, allocating storage, loading of programs, etc.
10. Software design - design of application programs and utility programs.
11. Storage device consideration - Decide which physical medium should contain the program modules, the files and the dialoque message.
12. Select proper block size for buffering - select primary storage block size for message buffers, file buffers, and relocatable or overlayable areas.
13. Decide the number of each type of buffer.
14. Establish storage requirements.
15. Develop model of system - to allow the study of the effects of various volumes and proportions of messages on the response time and throughput capabilities.

VII. CURRENT DEVELOPMENT OF TRANSACTION PROCESSING SYSTEMS

There are a number of transaction systems in operation today and the rate of applications are rapidly increasing. Many larger firms have TPS available as well as several smaller firms. Some of the better known TPS available for general use are TIP from Univac, CICS from IBM [8], TPE from Honeywell [7], TASK/MASTER from Turnkey Systems, INTERCOMM from Informatics, ENVIRON/1 from Cincom Systems, and SHADOW II from Cullinane Corporation.

As previously mentioned, applications are found in a vast variety of organizations from banking to government. The earliest TPS was the SABRE system developed for American Airlines for passenger flight inventory.

In the literature there are few detail documentations of TPS. Some military organizations use the TPE of the Honeywell system [5,14]. One large TPS described [4] is used by Time, Inc. that has 200 CRT terminals, a data base of five billion characters, processes 750,000 transactions per week, and provides a mean response time of one second.

VIII. PERFORMANCE EVALUATION OF TRANSACTION PROCESSING SYSTEMS

To date, there are essentially no documented studies on performance evaluation of TPS. Performance evaluation are commonly used in computer systems for design, selection, and tuning (peaking). A common and desired approach to use is the modeling approach. There are essentially no models for TPS in the literature. TPS are similar to time-sharing

system, but are different enough that these models are usually not applicable. This is a major area of TPS needing research.

IX. FUTURE RESEARCH NEEDS

As stated above, a major void exists in a lack of models and performance evaluation of TPS. This is not surprising in that other types of computer systems, e.g. batch and time-sharing, were developed and became operational before models of their behavior were developed for performance evaluation. For TPS, this is the next major step needed in their development.

Prior to development of models of TPS for design, selection, and tuning, an understanding of their operation is necessary. This means empirical studies must be made of operating TPS to obtain the necessary insights, data on their stochastic behavior, and their resource requirements. Certain behavior can be hypothesis, but must be determined if true. For example, are service requests approximately constant for each type of transaction? Are transaction arrivals Poisson? Are resource requirements similar for various types of transactions?

After some empirical studies have been performed, models for TPS can be developed. One just has to look at the large number of models developed of time-sharing systems to become aware of the fact that a large amount of research and development is necessary for model development of TPS.

X. SUMMARY

Transaction Processing Systems were described and pointed out to be a new type of computer system that is rapidly being developed and implemented. It was also noted that there are essentially no reported models or studies on performance evaluation of TPS and this is the next step needed in their development.

XI. BIBLIOGRAPHY

1. Beck, L., Automatic Design of Structured Data Processing Systems, Ph.D. Dissertation, Southern Methodist University, 1975.

2. Booth. G., Transaction Processing Systems, Data Management, July 1972.

3. Davenport, R.A., Design of Transaction-Oriented Systems Employing a Transaction Monitor, Proceedings ACM Annual Conference, November 1974.

4. Gerami, C.R., Shields, T.R. and Weiland, R.J., Transaction Queueing and Cylinder Logic Access in the Time, Inc. Magazine/Book/ Record System, AFIPS Conference Proceedings, 1976.

5. Gerke, S.P., Performance Projection and Evaluation for a Transaction-Oriented System, Symposium on the Simulation of Computer System II, 1974.

6. Hirchfeld, L.J., Design Methodology for Transaction Processing Systems, Ph.D. Dissertation, University of Pennsylvania, 1975.

7. Honeywell Information Systems, Series 600/600 GCOS Transaction Processing System User's Guide, Order No. DA82, 1973.

8. International Business Machines, Customer Information Control System (CICS) General Information Manual, GH20-1028-4.

9. Jessen, T.D., Transaction Oriented Minicomputer Allows Flexible Design of the Controlled Materials Information System, NTIS report UCRL-77948, 1976.

10. Lefkovitz, D., Data Management for On-Line Systems, Hayden Book Company, Inc., 1974.

11. Lefkovitz, D., and Hirchfeld, L.J., Transaction Processing via Processor Network, unpublished paper, Department of Computer Science, University of Pennsylvania, 1975.

12. McKee, D.J., How Transaction Cost Declines as Data Networks Get Larger, Data Communications Systems, September, 1973.

13. Osborn, R.A., Bain, W.P., Lloyd, T. and Perring, P.J., SPG - A Programming System for Commercial Transactions Processing, The Computer Journal, November 1975.

14. Rome Air Development Center, TPAP Operating System Executive, Technical Report, RADC-TR-75-212, AD # 016438

15. Schember, K., Optimal Design of Files for Transactions-Oriented Data Base Systems, Ph.D. Dissertation, Texas A & M University, 1976.

16. Stern, H.C., Cost/Benefit Analysis of Transaction Processing Systems, ACM Computer Science Conference, Washington, D.C., 1975.

17. Sticker, K., User Requirements on Operating Security of Transaction Oriented Electronic Data Processing Systems [in German] OUTPUT (GOLDACH) Vol. 3, No. 12, 1974.

18. Stubblefield, F.W. and Dimmler, D.G., Transaction Processing in the Common Node of a Distributed Function Laboratory Computer System, IEEE Transactions on Nuclear Science, February 1975.

19. Theirauf, R.J., Systems Analysis and Design of Real-Time Management Information System, Prentice-Hall, 1975.

Section 6

ANALYSIS AND DESIGN OF A COST-EFFECTIVE ASSOCIATIVE PROCESSOR FOR WEATHER COMPUTATIONS

WEI-TIH CHENG
Advanced Systems Development Division
International Business Machines
Yorktown Heights, N. Y. 10598

TSE-YUN FENG
Department of Electrical & Computer Engineering
Syracuse University
Syracuse, N. Y. 13210

Abstract-- An associative processor organization with application to weather computations, and specifically, to a two-level atmospheric general circulation model developed by Mintz and Arakawa is used as the application problem to be implemented for system analysis and design. These system parameter changes include the arithmetic-logic capabilities, the number of processing elements, different speeds, structures, and access techniques of the memory, addition of a high speed temporary storage, and the use of an auxiliary storage unit. Pertinent data are calculated for all these systems and a cost-effective system is finally determined after the comparisons of these analysis results are made. This final system is then evaluated and compared with the original system.

I. INTRODUCTION

Since it would be impossible for a system designer to expect a computer system to be cost-effective for all areas of applications, in this study an associative processor organization is selected to solve the weather computation problems. This is because associative processing provides a natural combination of arithmetic-logic and search-retrieval capabilities which is a desired characteristic for many mathematical problems, such as matrix operations, fast Fourier transform, and partial differential equation [1]. The solution of partial differential equations is essentially the basic mathematical tool for weather computations. Among many other weather problems, a two-level atmospheric general circulation model developed by Mintz and Arakawa [2] is chosen as one representative exam le for demonstrating the suitability of the associative parallel processor for such problems. The essential use of this general circulation model is as an experimental tool for studying the nonlinear behavior of the atmosphere. It is intended to use this model to explore the sensitivity and response of the world's climate to either deliberate or inadvertent modification, through the various settings of the initial and boundary

II. AN ASSOCIATIVE PROCESSOR ORGANIZATION

In order to make our investigation meaningful an associative processor organization is first assumed. The system consists mainly of five component [3] as shown in Figure 1. It is assumed that there are a total of 1024 PE's in this system. Each PE has a 256-bit associative word divided into four 32-bit fields (A, B, C, D), 16 bits of temporary storage (TS), an M bit, and one bit-serial arithmetic-logic unit. All 1024 M bits form a mask register which is used to activate or deactivate the corresponding PE's. Data words stored in the fields are fetched bis*-sequentially to be processed by the arithmetic-logic unit. The results are stored back to the designated destination.

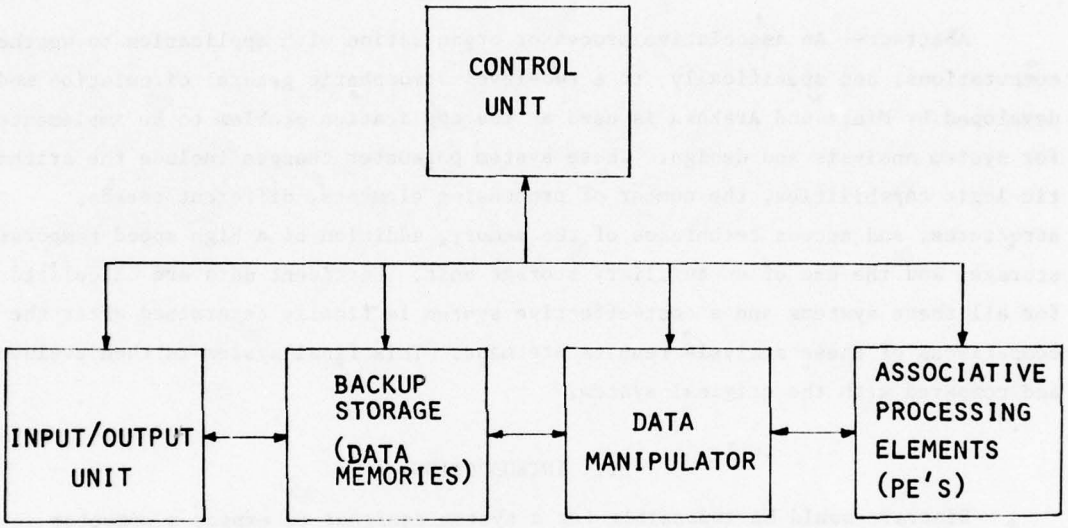


Figure 1. An associative parallel processor organization.

The data words are 32-bit long and represented in two's complement form throughout the system. Floating-point numbers have an 8-bit exponent and a 24-bit mantissa.

III. IMPLEMENTATION OF THE GENERAL CIRCULATION MODEL ON AP

The original program of the Mintz-Arakawa two-level atmospheric general circulation model was written in FORTRAN [2]. It has been translated into parallel processing programs [4]. The whole program is constructed mainly of two components.

*All ith bits of a given set of operands from the ith bit slice, or bis, of the set.

The first is the updating of variables in five time-steps by computing their time derivatives without the influence of the source terms. During each time-step the computing procedure is processed twice as required by the backward differencing scheme. We shall designate this computing procedure as PART I program which is executed in a total of ten times during the five time-steps. After the variables have been updated ten times through the five time-steps by PART I program, the effect of the source terms is added to the computation of the time derivatives. The part of program which produces the source terms is designated as PART II program.

PART I and PART II programs are segmented into 18 and 17 subprograms, respectively. These subprograms are executed one at a time in a sequential manner with each program. All these subprograms are used to update the basic variables of the general circulation model [2]. Each variable has a data set of the size of $46 \times 72 = 3312$. Since the capacity of the AP is assumed to be 1024, each of these data sets as well as other generated along the process, will have to be segmented into groups of proper size. For implementation details and other considerations see Reference [4].

IV. OPERATIONAL CHARACTERISTICS AND ANALYSIS OF THE GENERAL CIRCULATION MODEL

A. Operation Counts

Based upon the parallel processing programs we first obtain the operation counts of each subprogram. The distribution of operations is then analyzed.

Table 1 gives the operation counts for OVERALL program which includes the execution of PART I program ten times and the execution of PART II program once over the complete data sets.

B. Operation Distributions

From Table 1 we can see that arithmetic operations take up more than one third of all operations; data transfers among PE's take up more than one fifth; while load and store operations are about 15% and 5%, respectively.

Another type of operation distribution is obtained by calculating the distribution according to the system components involved in the operations. Two major categories of operations are operations involving both the backup storage and PE's and operations involving PE's alone (Table 2). Within each of these two categories of operations some operations require data manipulating functions and others do not. The total involvement of the data manipulator is also given in Table 2.

C. Estimation of Execution Time

The most fundamental operation executed in PE's to perform various arithmetic operations is the addition of two bises. This addition consists of the reading of two bises to be added, execution of the addition and the writing of the resultant conditions.

Table 1. Operation Counts and Execution Times of Overall Program

			Number of Operations	Execution Time per Operation (μsec)	Total Execution Time for OVERALL Program (μsec)
ARITHMETIC	Common	±	1528	126.7	193903.2
		×	1544	107.2	165516.8
	Field	±	8660	136.3	1180358.
		×	4354	207.4	903019.6
		/	1496	269.9	403770.4
	x,/ by 2^i		1616	2.4	3878.4
LOAD	Pattern	1	3888	20.8	80870.4
		2	696	49.75	34626
		3	1516	49.75	75421
		4	1880	49.75	93530
	One-Bis		292	0.65	189.8
STORE	Pattern	1	1066	27.225	29021.85
		2	292	59.775	17454.3
		3	904	50.95	46058.8
		4	440	59.775	26301
	One-Bis		28	0.85	23.8
TRANSFER	Direct		884	8.0	7072
	Shift		4424	16.1	71226.4
		M	2522	16.1	40604.2
	One-Bis		3504	0.25	876
TEMPORARY STORAGE			1423	27.2	38705.6
LOGIC			650	0.05	32.5
SET MASK			5054	0.10	252.7
MULTIPLICATE			660	28.9	170.4
MULTIPLE WRITE			1136	0.15	2547.2
SEARCH			398	6.4	2878.2
BIT SHIFT		24	468	6.15	670.8
		8	312	2.15	374.4
FIXED-POINT ADD		C	156	2.4	374.4
		F	156	2.4	3438802.15
TOTAL			51947	No. of Operations/sec = 15106	

sum bis into PE's. With the semiconductor type of PE associative memory (PEAM), two bises can be read out simultaneously, one in true form and the other in complement form. Thus a single bis addition takes 300 nsec, with the following timing assumptions for our system.

Operation:	Read PE	Write PE	Execute	Read backup storage	Write backup storage	Set DM control	DM delay
Time(n.sec):	100	150	50	500	750	100	250

Based on the above assumptions, the execution times for various operations and routines can be estimated. And according to the estimated execution time of the various basic operations we can calculate the total execution time for the OVERALL program. These are also given in Table 1. Notice that more than 80% of the total execution time is spent on arithmetic operations. A discussion on how to improve the total performance by speeding up the arithmetic operations will be given in the following section.

The execution time distribution of this system (System 1) according to system components can also be found in Table 2. Table 2 shows that only 13.41% of the total execution time is spent on executing operations involving both the PE's and the backup storage, the rest of the time is on operations executed in PE's alone. The operations which involve the data manipulator take up about 11.5% of the total execution time in processing the general circulation model.

D. Utilization and Efficiency of PE's

We define utilization, u, and efficiency, E, of PE's as follows:

$$u_i = \frac{\text{No. of PE's utilized in operation } i}{\text{Total No. of PE's available}} \qquad (1)$$

$$E = \sum_i u_i \cdot t_i/T = \sum_i u_i \cdot r_i \qquad (2)$$

where t_i is the execution time of the operation i, T is the total execution time of a program and r_i the percentage of t_i with respect to T.

From the parallel processing programs we see that almost all the arithmetic operations involve all the PE's where data are stored. This, however, does not give a 100% utilization. This is because the subprograms are executed four times for four groups of data of each complete data set and the first three groups of data occupy all the $14 \times 72 = 1008$ PE's but the fourth group utilizes only $4 \times 72 = 288$ PE's. Thus an average utilization for arithmetic operations is calculated as follows:

$$u_1 = \frac{1}{4} \times (1008 + 1008 + 1008 + 288) \times \frac{1}{1024}$$

$$= 0.8086$$

Table 2. Operation Counts and Execution Times According to System Components for OVERALL Program

		Operation		Execution (System 1)		Execution (System 10)	
		Count	%	Time (μs)	%	Time (μs)	%
Backup Storage ⟷ PE's	With DM	5547	10.68%	283470.85	8.24%	51985.30	4.38%
	Without DM	7536	14.51%	177887.45	5.17%	42980.15	3.62%
	TOTAL	13083	25.19%	561358.30	13.41%	94965.45	8.00%
PE's Alone	With DM	6946	13.37%	111830.60	3.25%	111830.60	9.42%
	Without DM	31918	61.44%	2865899.00	83.34%	980005.50	82.58%
	TOTAL	38864	74.81%	2977729.60	86.59%	1091836.10	92.00%
Total DM Involvement		12493	24.05%	495301.45	11.49%	163815.90	13.80%

The load and store operations have four different patterns and each pattern involves the loading or storing of different subsets of groups of data set. When different numbers of PE's are being loaded or stored in various patterns of operations, different average utilizations result. Utilizations for transfer operations can similarly be calculated. Table 3 includes all the utilizations for different operations.

Now that we have obtained the PE utilizations of the major operations we can calculate the PE efficiencies for processing PART I, PART II and OVERALL programs according to Eq. (2). Table 3 also gives the efficiencies of the three programs.

E. Relative Performance Measure

Evidently, the overall performance of the processor for the general circulation model programs cannot rely on the efficiency alone. At least two other important items should also be included in the consideration, these are the total execution time and the cost of the hardware. But for the moment, we shall concentrate only on the performance aspect of the system.

We now define the performance measure of a given system to be

$$P = \frac{E}{T^{\alpha}} \qquad (3)$$

where E is the efficiency of PE's, T is the execution time, and α is a weighting index of execution time with respect to efficiency. The reason that α is necessary is that a range of measurements with different degrees of emphasis on execution time with respect to efficiency can be made. Using the performance measure defined above we can do some comparative study of the overall performance evaluation of different systems. To achieve this we define a relative performance measure of System A with respect to System B, R_{AB}, to be

Table 3. PE Utilizations and Efficiencies

OPERATION			UTILIZATION u_i	PART I t_i/T	PART I $u_i t_i/(T \times 10^{-2})$	PART II t_i/T	PART II $u_i t_i/(T \times 10^{-2})$	OVERALL t_i/T	OVERALL $u_i t_i/(T \times 10^{-2})$
ARITHMETIC			0.8086	0.8031	64.9400	0.9266	74.9200	0.8290	670809
	Pattern	1	0.8086	0.0206	1.6670	0.0336	2.7150	0.0235	1.9020
		2	0.4922	0.0128	0.6300	0.0010	0.0500	0.0101	0.4960
		3	0.4922	0.0271	1.3340	0.0048	0.2370	0.0219	1.0880
		4	0.5223	0.0354	1.8480	0	0	0.0272	1.4210
	One-Bis		0.8086	0.00005	0.0040	0.00005	0.0040	0.00005	0.0860
STORE	Pattern	1	0.8086	0.0083	0.6750	0.0089	0.7170	0.0084	0.6820
		2	0.4621	0.0063	0.2930	0.0009	0.0420	0.0051	0.2350
		3	0.5391	0.0170	0.9140	0.0016	0.0840	0.0134	0.7220
		4	0.4621	0.0099	0.4600	0	0	0.0076	0.3530
	One-Bis		0.8086	0	0	0.00003	0.0020	0.0000	0.
TRANSFER	Direct		0.8086	0.0016	0.1270	0.0037	0.3000	0.0021	0.
	Shift		0.7598	0.0268	2.0360	0.0005	0.0380	0.0207	1.5730
		M	0.0130	0.0153	0.0200	0.0002	0.0003	0.0118	0.0150
	One-Bis		0.8086	0.0001	0.0100	0.0007	0.0560	0.0003	0.0310
EFFICIENCY $\Sigma u_i t_i/T$				0.7506		0.7919		0.7570	
OPERATION TOTAL $\Sigma t_i/T$				0.9843		0.9826		0.9812	
TOTAL EFFICIENCY $(\Sigma u_i t_i/T)/(\Sigma t_i/T)$				0.7626		0.8059		0.7715	

$$R_{AB} = \frac{P_A}{P_B}$$

$$= \frac{E_A}{E_B} \left(\frac{T_B}{T_A}\right)^{\alpha} \qquad (4)$$

which will be used repeatedly in comparing the relative merits of various systems.

V. SYSTEM VARIATIONS AND PERFORMANCE EVALUATIONS

In the preceding section the programs of the general circulation model were analyzed based upon the assumed AP organization and its capabilities presented in Section II. Now we shall examine how some of the system parameter changes affect the overall performance of the system processing these programs. The alternative systems with the following variations are presented:

1. four-bis-parallel arithmetic-logic capabilities,
2. 2048 PE's,
3. 4096 PE's,
4. new backup storage with slower speed,
5. new backup storage with faster speed,
6. modularized backup storage,
7. backup storage banks with interleaving capability, and
8. addition of a high speed temporary storage (HSTS).

We shall designate the system with the above system parameter changes as Systems 2, 3, ..., and 9, respectively, with the original system as System 1. For each of the alternative systems the following data are calculated for the processing of the general circulation model programs.

1. new execution time and speedup ratio when compared with the original system,
2. new PE utilizations and efficiencies, and
3. relative performance evaluation with respect to the original system.

A. Four-Bis-Parallel Arithmetic-Logic Capabilities

In the following paragraphs we consider a four-bit-parallel arithmetic unit for each PE.

If the four-bis-parallel capability is provided to the system, yet the data transfer rate is not increased accordingly, there cannot be any significant increase in overall processing speed. Therefore, we have to change the data path between PEAM and the arithmetic-logic unit so that four bises of data can be accessed simultaneously. This provision may be realized through the use of four identical associative memories with proper interconnections between them and the arithmetic-

logic unit.

The essential hardware requirement for four-bis-parallel add and subtract operations is a four-bis-parallel adder due to their simple iterative read-add(subtract)-write sequence. As for multiplications, not only there are many different algorithms, but also each algorithm requires different hardware setup. Since we are trying to obtain a balance between speed gain and hardware economy in providing four-bis-parallel arithmetic-logic operations, before we finalize the design of the arithmetic and logic unit, we examine some of the multiplication algorithms. Three algorithms are under consideration: direct multiplication scheme used by TI's SIMDA [5], Booth algorithm used in bis-sequential version of the processor, and polynomial algorithm used in Machado's proposed SPEAC system [6].

After examining and comparing these three multiplication algorithms, the polynomial algorithm is chosen due to its speed advantage and less storage requirements.

Based upon the four-bis-parallel capabilities in the arithmetic-logic unit of PE's, a complete set of algorithms for all the necessary arithmetic operations were developed [4]. Using the same timing assumptions as we used for bis-sequential operations we have the estimated execution time for the four-bis-parallel arithmetic operations (Table 4). Table 4 also compares the execution times of bis-sequential and four-bis-parallel arithmetic operations by giving the speedup factors.

Table 4. Execution Time of Floating-Point Arithmetic Operations (μsec)

Operation		Bis-Sequential	Four-Bis-Parallel	Speedup Factor
Add	Common	126.7	28.85	4.39
	Field	136.3	52.5	2.6
Subtract	Common	126.7	28.85	4.39
	Field	136.3	52.5	2.6
Multiply	Common	107.2	55	1.95
	Field	269.9	95.80	2.82
Divide	Common	253.9	95.60	2.66
	Field	269.9	95.80	2.82

With the new execution times for these various operations, we can calculate the subtotal and total execution times for each individual operations and the entire programs. The first row of Table 5 gives the speedup factors of PART I, PART II, and OVERALL programs when compared with the original system. We can see that the speed for processing the general circulation model on System 2 is more than doubled.

The utilization of PE's for all operations remain unchanged in System 2. The new PE's efficiencies, derived from the new execution time distribution are reduced

Table 5. Speedup Factors

SYSTEM	SPEEDUP FACTOR WITH RESPECT TO SYSTEM 1		
	PART I	PART II	OVERALL
2	2.12	2.66	2.22
3	2.05	2.00	2.04
4	4.26	4.02	4.08
5	0.91	0.95	0.92
6	1.06	1.03	1.05
7	1.04	0.99	1.03
8	1.14	1.05	1.12
9	1.06	1.02	1.05
10	2.86	3.04	2.90

as shown in second row of Table 6. This reduction results from the great decrement in execution time of arithmetic operations which have a high utilization of 0.8086.

Table 6. Comparison of PE Efficiencies

SYSTEM	EFFICIENCY		
	PART I	PART II	OVERALL
1	0.7626	0.8059	0.7715
2	0.7064	0.8008	0.7246
3	0.7757	0.8070	0.7830
4	0.7948	0.8083	0.7981
5	0.7467	0.8042	0.7594
6	0.7704	0.7065	0.7788
7	0.7748	0.8069	0.7825
8	0.7857	0.8078	0.7911
9	0.7722	0.8067	0.7803
10	0.7713	0.8063	0.7632

We can now calculate the relative performance of four-bis-parallel processing with respect to bis-sequential processing. R_{21} in Fig. 2 shows the relative performance between Systems 1 and 2 versus α. It indicates that if the improvement in execution time is more important than in PE efficiency, the four-bis-parallel arith-

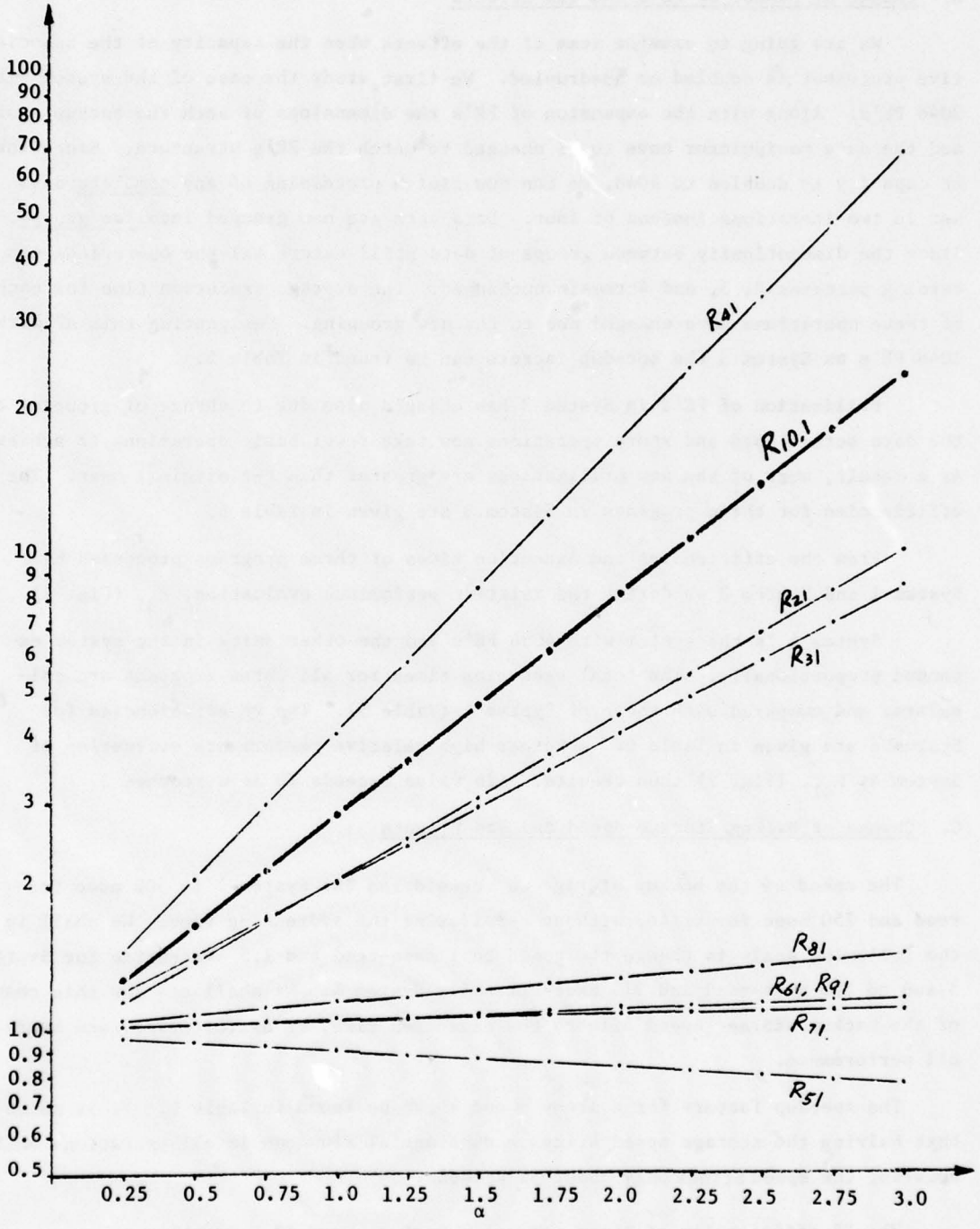


Figure 2. Relative performance evaluation.

metic-logic technique is more effective. Even when α is 1 (the efficiency and the execution time are of equal importance) the relative performance measure has a value of about 2.

B. Change of Number of PE's and Its Effects

We are going to examine some of the effects when the capacity of the associative processor is doubled or quadrupled. We first study the case of the system with 2048 PE's. Along with the expansion of PE's the dimensions of both the backup storage and the data manipulator have to be changed to match the PE's structure. Since the AP capacity is doubled to 2048, we can now finish processing of any complete data set in two iterations instead of four. Data sets are now grouped into two groups. Since the discontinuity between groups of data still exists all the operations concerning patterns 2, 3, and 4 remain unchanged. The average execution time for each of these operations have changed due to the new grouping. Designating this AP with 2048 PE's as System 3 the speedup factors can be found in Table 5.

Utilization of PE's in System 3 has changed also due to change of grouping of the data sets. Load and store operations now take fewer basic operations to achieve. As a result, most of the new utilizations are greater than the original ones. The efficiencies for three programs in System 3 are given in Table 6.

From the efficiencies and execution times of three programs processed by System 1 and System 3 we derive the relative perfomance evaluation, R_{31} (Fig. 2).

System 4 is the system with 4096 PE's and the other units in the system expanded proportionally. The total execution times for all three programs are calculated and compared with those of System 1 (Table 5). The PE efficiencies for System 4 are given in Table 6. A rather high relative performance evaluation of System 4, R_{41}, (Fig. 2) thus results. Its value exceeds 70 as α reaches 3.

C. Change of Backup Storage Speed and Its Effects

The speed of the backup storage we considered for System 1 is 500 nsec for read and 750 nsec for write, without considering the addressing time. We shall in the following analysis change the speed to 1 μsec-read and 1.5 μsec-write for System 5 and to 250 nsec-read and 375 nsec-write for System 6. We shall see how this change of the backup storage speed affects the execution time, PE efficiencies, and overall performance.

The speedup factors for Systems 5 and 6 can be found in Table 5. It is noted that halving the storage speed brings a substantial slowdown in all operations while doubling the speed brings only about 5% speedup.

The PE efficiencies of three programs, PART I, PART II, and OVERALL, processed in System 5 are lower than those in System 1 due to the increase of percentages in

load and store operations while the PE efficiencies for System 6 are increased (Table 6). The relative performance evaluations of Systems 5 and 6 are plotted in Fig. 2.

D. Modularization of Backup Storage and Its Effects

In Section III we have mentioned the discontinuity of data sets when they are stored in groups of blocks. The suggested means to solve this problem is the creation of new data sets to speed up the data transfer between the backup storage and PEAM. As a result of using these new data sets a 24% speedup is computed for all the load operations performed in PART I program of the general circulation model.

An alternative way to speed up the loading of Patterns 2, 3, and 4 data sets is to modularize the backup storage. The backup storage has Q modules with equal dimensions. Their vertical dimensions is the same as that of the data manipulator and also the number of PE's. A mask register is associated with each storage module in order to select or "screen" the words to be read out or written into. At most one ith word of all the storage modules can be read out or written into at a time. The words thus read out from any number of modules can then be manipulated to be loaded in a desired format into PEAM. Storing data from the PEAM into the backup storage modules goes through the same procedure in reverse order.

All other operations are executed in exactly the same way as in System 1. The speed comparison between System 1 and this system with the modularized backup storage, designated as System 7, is given in Table 5. The consequent changes in execution-time distributions contribute to slightly increase in PE efficiencies (Table 6). The relative evaluation of System 7 is plotted in Fig. 2.

E. Backup Storage Banks with Interleaving Features and Its Effects

In using an interleaving system to process the general circulation model, we do not change either the data structure or the programs used in System 1. It affects the execution time of load, store, multiplicate and temporary store operations.

The total execution times for this system, System 8, are calculated and compared with those of System 1 (Table 5). From Table 5 we see that the improvement in total execution time in this system with interleaving BSB's is an excellent 12% over System 1, even though the combined execution time of load, store, temporary store, and multiplicate operations take only 13.425% in the original system for the OVERALL program. This speedup is also reflected in the calculation of PE efficiencies. As the data transfer type of operations take less portion of the total execution time to perform, the arithmetic operations which have higher PE utilization now take larger percentage of the total time. This boosts the PE efficiencies (Table 6). The plot of the relative performance evaluation of System 8 with re-

spect to System 1, R_{81}, can be found in Fig. 2.

F. Addition of A High Speed Temporary Storage and Its Effects

Here we present a system with a separate storage unit, a high-apeed temporary storage (HSTS), used to handle the temporary store operations and also used as a buffer between the backup storage (through the data manipulator or bypassing it) and PEAM. The speed of this high-speed temporary storage is compatible to that of the PE.

The comparison of subtotal and total execution times between System 1 and System 9 is given in Table 5. Note that the speed improvement of the OVERALL program in System 9 over System 1 is about 5%, a very significant gain, considering that the number of operations with improved execution time by the use of HSTS is only 12.02% of all the operations.

PE efficiencies are increased in System 9 as compared to System 1 because the execution times in performing load and store operation are reduced (Table 6). The relative performance evaluation of System 9 with respect to System 1, R_{91}, does not have large values as shown in Fig. 2.

G. Storage Requirement and Auxiliary Storage Unit

When every data set is too large in size to be processed at one time as in this weather computation problem with the assumed organization, there are two kinds of processing schemes: the complete data set processing scheme and the complete program processing scheme. The complete data set processing scheme is such a design of program flow that every data set is completely processed by each subprogram before going through the next subprogram. On the other hand, a complete program processing scheme is so designed that a segment (i.e., group) of the data set is processed by the entire program before the next segment of data set is processed. The result of a detailed analysis of all subprograms and their storage requirements for both schemes is summarized in Table 7.

From Table 7 we see that the sizes of the backup storage required to process the general circulation model are 160K and 79K in the complete data set processing scheme and the complete program processing scheme, respectively, without the use of the temporary storage. When the size of the grid system becomes larger, say, doubled in both horizontal and vertical dimension, the size of the backup storage increases rapidly (277K and 115K) and its access speed may be slowed down considerably. The use of an auxiliary storage unit is thus considered.

An APL simulation program [4] is written to simulate the data transfer activities between the backup storage and the disk. Various sizes of the backup storage are used in the simulation program. From the simulation we see the rapid decrease in the number of roll-in and roll-out operations required as the size of the backup

Table 7. Storage Requirement for the General Circulation Model

PROCESSING SCHEME \ STORAGE	ONE STORAGE	SEPARATE STORAGE		
	Backup Storage	Backup Storage	Temporary Storage	Total
Complete Data Set Processing Scheme	160K	156K	10K	166K
Complete Program Processing Scheme	79K	75K	10K	85K

storage increases, especially when the size of the backup storage is small. When the size of the backup storage reaches 43K words, 48 roll-in and 36 roll-out operations are required and they remain constant for larger backup storage sizes. This total of 84 roll-in and roll-out operations account for the roll-ins and roll-outs of the twelve residing data sets at the beginning and the end of each updating cycle, respectively. Thus, if an extra 36K words capacity is added to the backup storage no roll-in and roll-out operations are necessary. This agrees with the backup storage requirement for the general circulation model when processed by the complete program processing scheme, namely, a maximum of 79K words are required for the backup storage without the use of the auxiliary storage unit.

VI. A MORE COST-EFFECTIVE ASSOCIATIVE PROCESSOR SYSTEM

In Section V we have presented eight different systems and showed, in each of them, how the system parameter change affects the overall performance. However, each of these system parameter changes was concentrated in only one component of the entire system. Three of these eight system variations were on the associative processing elements and five on the backup storage. In this section we try to compare the advantages and disadvantages of these systems and then combine the merits of these systems in order to arrive at a more cost-effective system than the assumed system presented in Section II.

A. Comparison of the Cost-Effectiveness of Alternative Systems

1. Associative Processing Elements

Three systems, Systems 2, 3, and 4 mentioned in Section V dealt with alterations of the capability and the number of the associative processing elements in the system. Among these the four-bis-parallel processing capability affects only the

hardware of PE's leaving the rest of the system intact. But in the cases of the other two alternatives, the sizes of other components in the system change along with the size change of the associative processing elements. A cost analysis on these two alternatives [4] show that System 3, with 2048 PE's, and System 4, with 4096 PE's, have, respectively, a cost of double and quadruple of that of System 1 and the overall cost of System 2 is less than double of that for System 1.

Thus, from the cost analysis and the speedup factors given in Table 5 for Systems 2, 3, and 4, it seems that System 2 is most favorable. Table 6 indicates that the PE efficiencies for Systems 3, and 4 are a little higher than that for System 2 in processing this weather problem, but in applications where the sizes of the data sets being processed are not as large as the capacity of PE's the efficiency will become much smaller. In short, System 2 is relatively more cost-effective.

2. The Backup Storage Unit

There are five variations on the speed, the capability, and the structure of the backup storage unit, resulting in five different systems, Systems 5, 6, 7, 8, and 9. In considering the cost factors of these systems in comparison with System 1, it is apparent that System 5 is the only system which is less costly than System 1 because it has slower speed. System 7 and 8 have basically the same memory modules but with different accessory equipments such as masking control in System 7 and interleaving control in System 8. System 9 has the largest storage requirement in terms of overall storage size. If we process the general circulation model programs in a complete program processing scheme, Systems 5, 6, 7, and 8 would need only a total of 79K words while System 9 would need 75K words of backup storage and 10K words of HSTS (Table 7). With the speed of HSTS compatible to that of the associative memory System 9 evidently is much more costly than the others.

It may be difficult for us to have a more precise cost comparison among these systems without going into implementation details. On the other hand, it may not be necessary for us to have a more precise cost comparison to determine which of these systems is the most cost-effective one. From Tables 5 and 6 we can see that System 8 demonstrates the best performance in execution time in processing the general circulation model and also it improves the PE efficiency from 0.7715 in the original system to 0.7911. It seems that despite the difficulty in having a very precise absolute-cost comparison among Systems 5, 6, 7, 8, and 9, it is reasonable to choose System 8 for the improvement of the backup storage over the origina system.

B. Examination of a Cost-Effective System

We shall now examine how the new system, designated as System 10, with the combined improvements from System 2 and System 8, performs in implementing the general circulation model. The operation distribution remains the same as in System 1 since there is no change in programming. The total execution times for PART I,

PART II, and OVERALL program are 92833.94 μsec, 258462.15 μsec, and 1186801.5 μsec, representing speedup ratios over System 1 of 2.86, 3.04, and 2.90, respectively (Table 5).

We may look at the execution time from another point of view. As done in Section IV, total execution times are analyzed according to the utilization of the major components in the system. The associative processing elements are kept busy in all the operations so it has a 100% component utilization. Only 8.00% of the total time is spent on the data transfer operation between PE's and the backup storage, either through the data manipulator or bypassing it (Table 2). Again, the decrease in this figure from that for System 1 is due to a tremendous speedup in data transfers by interleaving the bises when transferred in and out of the backup storage. This low utilization of the backup storage suggests a need for further improvement in component utilization. To achieve this data sets to be processed in the associative processing elements may be stored at different levels of the memory hierarchy originally and can be brought to the unit physically right next to the PE's. The time required in the more complicated data management can be overlapped with the PE processing time. Even some of the data manipulating functions can be performed either before the data sets are actually being loaded into the PEAM or if necessary, after they are outputted from PEAM and before they are stored back into the storage. More sophisticated controls and detection of special features of the instruction sequences should be provided for this more efficient processing technique in order to prevent any type of conflicts among the components.

Table 2 also shows a quite significant increase, about 20%, in the use of the data manipulator when compared with System 1. The new PE efficiencies in processing the three programs of the general circulation model on System 10 are 0.7713, 0.8063, and 0.7632, respectively (Table 6). A slight decrease in efficiency is evident in PART I and OVERALL programs where the percentages of execution time in arithmetic operations (having a high PE utilization of 0.8086) dropped. The relative performance evaluation $R_{10,1}$ of System 10 with respect to System 1 ranges from 1.29 to a very respectable 23.88 as α varies from 0.25 to 3.0 (Fig. 2).

The efficiencies of the backup storage and the data manipulator for Systems 1 and 10 are summarized in Table 8. The efficiencies of System 10 are improved substantially over those of System 1.

C. Further Consideration of a Cost-Effective System

Consider that a disk unit is added to the system and dual control units are provided so that concurrent operation both in the PE's and in the backup storage and the disk unit can be performed. Adding this disk unit has twofold advantages: the utilization of the backup storage is improved and the size of the backup storage can

Table 8. Efficiencies of Backup Storage and Data Manipulator in System 1 and System 10.

SYSTEM COMPONENT	SYSTEM	PART I	PART II	OVERALL
BACKUP STORAGE	1	0.5972	0.7672	0.6153
	10	0.6104	0.7751	0.6303
DATA MANIPULATOR	1	0.4546	0.4112	0.4539
	10	0.4746	0.4376	0.4743

be reduced. From the result of Section V we have already known the number of roll-in and roll-out operations required for the system with certain size of the backup storage to process the general circulation model program. Our attempt here is to find the optimal size of the backup storage in the most efficient system.

Let us assume that an Alpha Data 16" disk unit is used. It has an average latency time of 16.8 msec and a flow rate of 10^7 bit/sec approximately. It takes about 3.2 msec to perform a roll-in or roll-out operation. The approximate time required to accomplish all the roll-in or roll-out operations is then the number of roll-in and roll-out operations times 20 msec. The optimal size of the backup storage should be such that the roll-in and roll-out operations do not increase the total execution time. That is, the roll-in and roll-out operations should overlap as much as possible with the time when the backup storage is idle. From Table 2 there is about 1120 msec during which period the backup storage is idle. Thus the maximum number of roll-in and roll-out operations is 1120/20 = 56. By checking Fig. 4 we find out that the minimum number of roll-in and roll-out operations is 84 when all 12 residing variables data sets (48K words) are stored on disk and the backup storage has a size of 43K words. If we add extra 12K words to the backup storage for three residing variables and the rest nine remain stored on the disk then the number of roll-in and roll-out operations is exactly 56. Thus, for implementing the general circulation model on such a system with four-bis-parallel capabilities and interleaving backup storage banks, the optimal size of the backup storage is 43K + 12K = 55K, provided that the complete program processing scheme is used.

The reduction of the size of the backup storage from 79K words in a system without a disk unit to 55K words, and about 30% reduction, may not be significant in terms of hardware cost. But when the grid system used in the general circulation model becomes denser, say, the numbers of grid points on the latitudinal and the longitudinal lines are doubled there could be much greater savings in the backup

storage.

D. A Cost-Effective Associative Processor Organization

In summarizing the result of the analyses, a final system, is presented here. Figure 3 gives a more detailed diagram of the optimal system. The descriptions of the system components are given below:

1. Control Unit: This unit stores the main programs, executes the instruction sequence, performs some sequential arithmetic operations and controls other components in the system by sending out appropriate control signals to them. Dual control capability must be provided in this unit so that the concurrent operations of data transfer between the backup storage banks and the disk unit, and executing arithmetic, search, or other operations in PE's alone can be successfully achieved. This dual control units supervise all the activities and prevent any conflicts from occurring.

2. Input/Output Unit: A disk unit with an average latency time of 16.8 msec is used as the auxiliary storage unit. Input/Output unit receives the commands from the control unit and performs data transfers between the disk unit and the backup storage without sacrificing the total execution time.

3. Interleaving Backup Storage Banks: There are four backup storage banks, each having a size of 14K words. When reading or writing data bises out of or into the backup storage banks in an interleaving manner the effective read and write times are 150 nsec/bis and 200 nsec/bis, respectively. This provision increases the overall system performance to a quite substantial extent. When the total execution time for processing the general circulation model programs is considered, it has an improvement of about 12%. When PE's are processing some data sets without the use of this component, the backup storage banks can load and unload data sets from and to the disk. This kind of concurrent operation increases the utilization of the system components and thus improves the overall system efficiency.

4. Data Manipulator: In order to have an efficient parallel processor some kind of data manipulating unit must be included in the system otherwise the time spent on preparing data sets to be processed may well offset the time saved by parallel processing. The separate data manipulator provided in this system not only speeds up data preparation but also adds more flexibility to the choice of data source and destination. A very competent set of data manipulating functions are built into this unit to enhance its performance. A total of about 14% of the total execution time is spent on operations involving this data manipulator when the general circulation model is implemented on this system. Shift and multiplicate operations are greatly used in this weather problem.

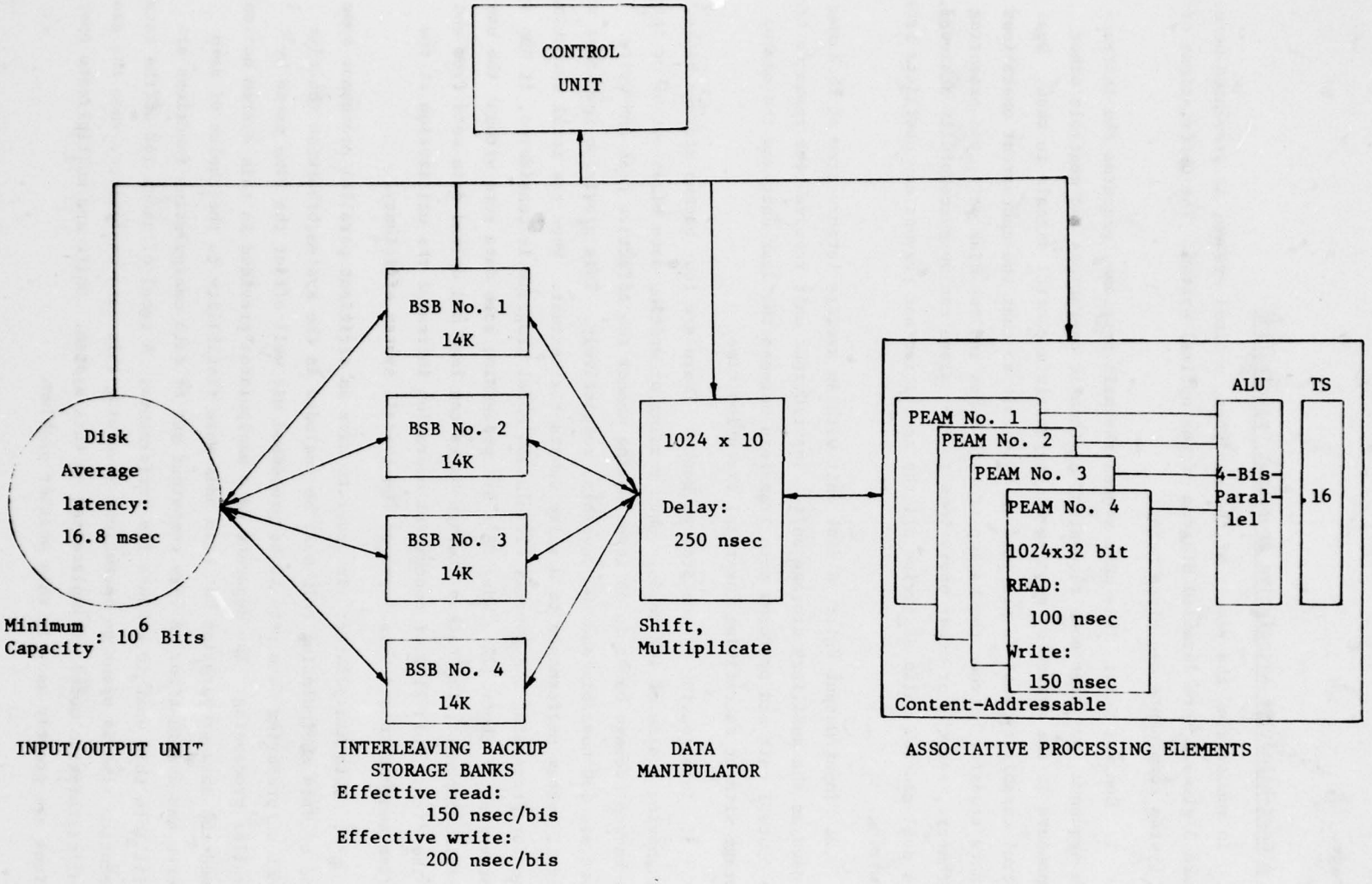


Figure 3. A cost-effective associative processor organization for weather computations.

5. Associative Processing Elements: After the analysis on the execution time speedup ratios it is evident that it is very advantageous to equip the PE's with four-bis-parallel processing capabilities. The overall speed improvement is more than 100% as a result of this change. The processing elements associative memory (PEAM) can be implemented by four smaller PEAM's each with 1K words. The data sets are stored in these PEAM's. One bis of data is accessed from each of the four PEAM's simultaneously to be processed by the arithmetic-logic unit. The read and write time of these PEAM's are 100 nsec/bis or word and 150 nsec/bis or word, respectively. There are 16 bises of TS for temporarily storing mask conditions, results of logic or search operations.

VII CONCLUSIONS

A cost-effective system for weather computations was determined after some comparisons were made among all the system alternatives. The comparisons included the cost factors (in terms of hardware complexity), total execution time, and PE efficiency. It has 1024 PE's equipped with four-bis-parallel processing capabilities and content-addressability, a separate data manipulator, four interleaving backup storage banks, a control unit with dual control capability and a disk unit. The implementation of the general circulation model on this optimal system was then analyzed. This system was again compared against the original system. The total throughput is increased by almost three times. A system efficiency evaluation which included the utilizations and efficiencies of all three major system components, the cost factor, and the total execution time was made for the two systems. There is about 10% increase in system efficiency in the optimal system.

The whole project required very tedious and laborious work of studying thoroughly the particular application problem and its program, rewritting the program into parallel processing program, gathering operational statistics, developing detailed algorithms for arithmetic operation (both bis-sequential and four-bis-parallel), analyzing, evaluating, and comparing system performance, and finally determining an optimal system. The completeness of the original document of the general circulation model facilitated the validity of this undertaking.

REFERENCES

[1] W. T. Cheng and T. Feng, Associative Computations of Some Mathematical Problems, Technical Report, RADC-TR-73-229, Rome Air Development Center, Rome, New York, August 1973, 76 pp, AD # 768978/9

[2] W. L. Gates, E. S. Batten, A. B. Kahle, and A. B. Nelson, A Documentation of the Mintz-Arakawa Two-Level Atmospheric General Circulation Model, R-877-ARPA, December 1971, Rand Corp., 408 pp.

[3] T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implementations", IEEE Trans. on Computers, Vol. C-23, March 1974, pp. 309-318.

[4] W. T. Cheng, "A Cost-Effective Associative Processor with Application to Weather Computations," Ph.D. Dissertation, Syracuse University, Syracuse, New York, July, 1974.

[5] SIMDA, Single Instruction Multiple Data Array Processor Programmer's Manual, Handbook No. HB55-A72, Texas Instruments, Inc., October 1972.

[6] N. C. Machado, An Array Processor with A Large Number of Processing Elements, CAC-25, University of Illinois at Urbana-Champaign, January 1972.

Section 7

AAPL: AN ARRAY PROCESSING LANGUAGE

JOHN G. MARZOLF
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, New York 13210

Abstract -- AAPL is a dialect of APL designed to be more amenable to compilation than the standard version of APL. The principal differences include the use of name prefixes, the ability to accept a limited character set for denoting the primitive functions, some variations and restrictions on the use of the program-branching primitive, and some additional I/O primitives. The reasons for each of these modifications are discussed in detail, as well as the implications for transportability between the two dialects. An implementation of AAPL has been undertaken for the STARAN Associative Processor. An outline of this implementation and a progress report on the work is presented.

INTRODUCTION

APL as currently defined by the APL/360 implementation (see [1] and [2]) seems to be a natural candidate for a high-level language for SIMD-type parallel processors [3]-[5]. For a number of reasons, however, it does not appear that it can be conveniently compiled. For instance, the context-sensitive nature of a number of primitive symbols leaves their semantics ambiguous until execution time. Specifically, there are a group of primitive symbols which are used to denote two completely different functions depending on the number of arguments supplied with the symbol.

An example would be the cross (ordinary multiplication sign), which can be used monadically (i.e., with only a right argument) to represent the signum function, or it can be used dyadically (i.e., with both a left and right argument) to denote the multiplication function. In any given context the precise function intended can be determined by examining the source code to the left of the symbol. If there is some primitive scalar function on the left, then the cross has no left argument and hence is the signum function. If there is a right parenthesis or a number to the left of the cross, then it is known to be dyadic and thus is the multiplication function. For these cases the meaning is uniquely determined when the source code is entered. Such an arrangement allows compilation, provided one omits consideration of the memory management required for storage of the result.

The complication sets in when a name or identifier appears to the left of the

cross. There is no way of knowing, prior to the actual execution of the statement, whether this name will represent (at execution time) a variable value or a monadic defined function call - and that makes all the difference. Another difficulty in compiling APL stems from the absence of declaration statements in the language. This makes memory management a very complicated process.

One obvious response to these difficulties is to run APL interpretatively. Not only does this solve the problems mentioned, but it also buys a whole host of little extras that are very convenient for interactive programming. On the other hand, it entails a certain loss of execution efficiency. This is counter-productive to the increased computational throughput expected from parallel processing.

The solution that is proposed here is to introduce a dialect of APL (to be called AAPL, for 'An Array Processing Language') which can be more readily compiled, yet will remain as close as possible to the basic structure of standard APL. This permits parallel processing programs to be interactively written, tested and debugged on existing APL systems, and then transferred to the parallel processing systems for production use. It also provides the parallel processing programmers with a tested set of array oriented algorithms that have been developed by the APL programmers.

THE DESIGN OF AAPL

There appear to be two solutions to the problem of using the same symbol to represent two different functions, namely (1) change one of the symbols, or (2) provide naming conventions which explicitly identify the class of a name used on the left of the ambiguous symbol. The first solution is the obvious one, but it adds the additional burden of having to provide more primitive symbols - perhaps to the extent of creating a symbol set that will become unwieldy. This objection could be removed by a variation employing reserved words for certain of the primitive functions. In this case there is the more serious objection that it would be difficult to transport programs between the two dialects of APL. For this reason the second of the proposed solutions was chosen for AAPL. This also solves the related problem of detecting the syntax error in an attempted assignment of a value to a niladic function name or a label.

All names in AAPL are given alphabetic prefixes which specify the name-class of the object identified. There are five name-classes, and each is distinguished by a two letter prefix as shown in Table I. This means that all AAPL names are a minimum of three characters long. The first two must be a proper prefix and the remaining characters should be chosen according to the ordinary naming rules of APL. Such a procedure causes AAPL names to form a subset of APL names. Provided the names used in standard APL programs are chosen from that subset, all programs can be interchanged without alteration between the two dialects.

The selection of a two letter prefix scheme (rather than a one letter arrangement) was related to the desire of freeing AAPL from the I/O device restrictions imposed by the ordinary APL symbol set. In discussing this problem it should be noted that AAPL is intended to be a dialect of APL, and hence should differ from it as little as possible. On these grounds it was decided that the AAPL symbol graphics would be those of standard APL. However, the AAPL compiler would be designed to accept mnemonics as well as the standard symbols, since this would broaden the base of I/O devices that could be used with the system.

There have already been a number of proposals for such a set of mnemonics [6]-[8], but they generally employ a preceding escape character to differentiate the mnemonics from a similarly constructed APL name. In addition, the escape sequence is designed to be processed by the system input routine. This means that the conversion from the supplied mnemonic to the proper internal representation is made wherever the escape convention is encountered. In other words, if the escape sequence were used within a character literal, it would be converted by the system to the appropriate internal representation of the graphic for which it was the mnemonic. This is a suitable arrangement provided one is willing to allow the internal character count of some arrays to be different from that of their graphic representations. This is an anomaly which is avoided in AAPL.

There are no mnemonic escape characters used in AAPL. If the input is coming from a device capable of generating only a subset of the APL symbols, then those are the only symbols the device can enter into the system. What AAPL does permit, and which is not permitted in standard APL, is an alternate way of indicating to the compiler what precise computation is intended by the programmer.

The AAPL programmer may replace many of the standard APL graphics with a two letter mnemonic, which, because of the prefixing scheme chosen, cannot be confused with a name. When these mnemonics are entered from any device (including a device capable of entering the full APL symbol set) they are entered into the system precisely as the symbols for which they are the graphics (not as the symbols for which they are the mnemonics). The difference comes when the AAPL compiler encounters them. At this point they are translated into the proper execution code according to their mnemonic meaning.

For this reason not every APL graphic has a mnemonic representation, but only those that denote an executable operation, or are used as compiler directives, such as the lamp to indicate a remark. The mnemonics recognized by the AAPL compiler are shown in Table II. They are all two letter mnemonics, except for the letter N which is used to denote the hi-minus. This is treated as an exception since it is used as part of a numeric representation to denote a negative value, and is analogous to the use of the letter E in exponential format. In addition, the ASCII circumflex and the PL/1 logical NOT (hooked-minus) are considered as stylized versions of the hi-minus.

In cases where two mnemonics are shown for a given primitive, they are so

provided to improve the readability of the mnemonic source code, but they may be used interchangeably. The intended meaning is determined by the compiler following the same procedures it would use if the primitive graphic had been employed.

Using this set of mnemonics, the only symbols required to write any AAPL function would be the letters, digits, parentheses, period, quote, and space. This permits AAPL programs to be entered from virtually any I/O device. The only features not provided by the mnemonics are the first-axis functions (for rotate, compress, expand, reduction, and scan), and the trace and stop control vectors. The omission of the first axis functions is not serious since they may be obtained from the indexed versions of these functions. The trace and stop control facilities are debugging aids, and it was not considered necessary to include them in a production oriented system.

Three new primitive functions have been added to allow the full use of all available I/O devices. These primitives are not assigned graphics from the standard APL symbol set so as not to conflict with any future assignments. They are available only as mnemonics. There is one input function (IN) which is monadic and takes a two component vector for its right argument. This argument specifies a file number and a component number which is the logical identifier of a physical device and a specific record. The assignment of a logical identifier to a given physical file is made by one of the I-beam functions. This also opens the file. The result generated by the input function is the value of the data object identified by its right argument.

There are two functions provided for generalized output; OU is the mnemonic for bare output and OT denotes terminated output. These functions are identical except for a final new-line character (carriage return plus line feed) that is supplied by the system at the end of each OT output request. Both functions are dyadic. The left argument specifies a logical file and component (similar to the file/component specification of the input function) and the right argument provides the data object to be output. Like other APL primitives, the output functions generate a result, which in this case is the value of the right argument. In other words, ignoring the physical output produced, these functions act as if they were the identity function. An alternate way of viewing these two functions is as a specialized case of the assignment function where the left argument is the logical name of a shared variable [9].

The problem of memory management in the absence of dimension declarations is solved by subterfuge. Although subsequent versions of AAPL may wish to attack this problem directly, the present version solves it by a get-space routine that does the job at execution time. This means that there is not a unique correspondence between the variable names and their physical memory addresses. Data values are located through a data descriptor table. Once some experience is gained with this imple-

mentation it will be possible to determine if a better solution to the problem is demanded.

There has been an improvement in the efficiency of the branching mechanism supplied with standard APL. Since the whole question of a proper set of control structures for APL is under investigation (see, for example, [10] and [11]) the arrangement described here should be seen as only an interim solution. Although the GOTO is not eliminated (indeed, it remains the kernel of the whole control system) it is made subject to a number of restrictions designed to provide more efficient execution. As a by-product these restrictions significantly improve one's ability to trace the flow of control through a program.

There are only three types of syntax allowed when using the branch arrow. It may be used

(1) with a single label or the number 0,

(2) with an indexed label list, or

(3) with a single label or the number 0, followed by the mnemonic IF, followed by any AAPL expression that evaluates to a 1 or 0.

This means that the target(s) of any branch is (are) always visible at the source level. Although this is more restrictive than the ordinary APL usage, it avoids the possibility of a variable name or function name completely obscuring the destination of the branch.

The indexed label list mentioned in the second type of branch is constructed by simply juxtaposing a number of labels (or the number 0) as if they were the elements of a numeric vector, and then indexing this string. This is the format required by the AAPL compiler, but it is not permitted in APL. Nonetheless, its semantic equivalent may be easily obtained in APL by catenating the labels, enclosing them in parentheses and indexing the resulting expression. This multiple branch provides the "case selection" capability used by structured programming languages to execute one of a number of alternate program blocks.

The third type of branch uses a new primitive (IF) which is nothing more than the compression function with its arguments reversed, together with the restriction that its left argument must be a single label or the number 0. Although this function is not available in standard APL as a primitive, it is easily introduced as a user defined function. Indeed, there are no real transportability problems between the two dialects because of these branching restrictions. All that is required is that these restdictions also be followed in the standard APL programs.

THE IMPLEMENTATION OF AAPL

An implementation of AAPL is in progress for the STARAN Associative Processor at Rome Air Development Center. The AAPL compiler is itself written in AAPL. It

communicates with the I/O devices through a supervisor module written in MACRO-11 assembly code. This module runs on a PDP-11 which acts as the sequential control unit for STARAN. The supervisor module uses the basic I/O facilities of the DOS-11 batch operating system, and presents the AAPL compiler with a source program in a form called PCODE. This is a print-code, and is merely an extension of ASCII to allow for the APL graphics, including the overstrikes.

The AAPL compiler translates the PCODE source program into an object module in a form called QCODE. If the source program is a function definition, then the object module is merely stored in the control memory. Otherwise the compiler acts as a load-and-go system, deleting the object module when execution is complete.

QCODE is the AAPL execution code, and is basically a form of threaded code [12]. The QCODE machine is a stack oriented virtual machine composed of two main modules, one running on the sequential processor (SP) and the other on the associative processor (AP). There is also an interface module to manage the interprocessor communications via the interrupt system and shared memory (which is actually AP control memory that is accessible to SP).

The system attempts to overlap as much as possible of the SP and AP execution. The SP module fetches the object code, decodes it, performs the necessary housekeeping, and makes a call to AP for an array operation. While AP is satisfying this request, SP proceeds to the next object code.

The SP module consists of about 100 routines, of which about 70 have been written and tested. These routines have all been written in MACRO-11. The AP module is written in APPLE, which is the assembly language for the STARAN associative processor. Only a few of these routines have been completed.

The initial tests on the system have used a sequential simulation in place of the AP module. While it is not possible at this time to provide detailed performance statistics, preliminary tests using a stand-alone PDP-11/45 are extremely encouraging. For instance, the AAPL branch outperforms the APL branch (running on an IBM 370/155) by better than 4-1. This is an actual elapsed time measurement without an adjustment for the different intrinsic speeds of the two machines.

TABLE I. NAME PREFIXES

Prefix	Name Class
VA	Variable
LA	Label
NF	Niladic function
MF	Monadic function
DF	Dyadic function

TABLE II. AAPL MNEMONICS

Graphic	Name	Mnemonic	Graphic	Name	Mnemonic
+	Plus	PL	↑	Take	TA
-	Minus	MI	↓	Drop	DR
×	Times (Signum)	TI,SG	⍋	Grade Up	GU
÷	Divide	DV	⍒	Grade Down	GD
*	Exponentiate	XP	/	Compression (Reduction)	CM,RD
⌈	Maximum (Ceiling)	MX,CE	\	Expansion (Scan)	XN,SC
⌊	Minimum (Floor)	MN,FL	⊥	Base Value	BV
|	Residue (Absolute Value)	RS,AB	⊤	Representation	RP
⍟	Logarithm	LG	⍎	Execute/Evaluate	EX
!	Factorial(Binomial Coef)	FA,BC	⍕	Format	FM
○	Circular Fns (Pi Times)	CI,PI	⌹	Matrix Divide	MD
<	Less than	LT	←	Assign/Is	IS
≤	Less than or Equal	LE	[	Left Bracket	LB
=	Equal	EQ	]	Right Bracket	RB
≥	Greater than or Equal	GE	;	Semicolon	SM
>	Greater than	GT	I	I-Beam	IB
≠	Not Equal	NE	⍞	Quad	QD
∧	And	AN	⍠	Quote-Quad	QQ
∨	Or	OR		Input	IN
⍲	Nand	NA		Output-Bare	OU
⍱	Nor	NR		Output-Terminated	OT
~	Logical Not	NT	∘	Null/Outer Product	OP
?	Question Mark	QU	∇	Del	DL
,	Catenate (Ravel)	CA,RA	⍛	Locked Del	LD
⍴	Rho	RH	→	Go To	GO
⍳	Iota	IO		IF	IF
∈	Epsilon	EP	:	Colon	CL
⍉	Transpose	TR	⍝	Lamp/Remark	RM
⌽	Rotate	RT	¯	Hi-minus	N

REFERENCES

[1] APL/360-OS and APL/360-DOS User's Manual, IBM Publication GH20-0906 (1970).

[2] Sandra Pakin, APL-360 Reference Manual, Science Research Associates, Inc., (1972), 192 pp.

[3] Kenneth J. Thurber and John W. Myrna, "System Design of a Cellular APL Computer", IEEE Transactions on Computers (April, 1970), vol. C-19, No. 4, pp. 199-212.

[4] A. Hassitt, J. W. Lageschulte, and L. E. Lyon, "Implementation of a High Level Language Machine," Communications of the ACM (April, 1973), vol. 16, No. 4, pp. 291-303.

[5] A. D. Falkoff and K. E. Iverson, "The Design of APL", IBM J. Res. Develop. (July, 1973), vol. 17, pp. 324-334.

[6] Tom McMurchie, "A Limited Character APL Symbolism", APL Quote-Quad (November, 1970), vol. 2, No. 4, pp. 3-4.

[7] P. E. Hagerty, "An APL Symbol Set for Model 35 Teletypes", APL Quote-Quad (September, 1970), vol. 2, No. 3, pp. 6-8.

[8] Glen Seeds, "APL Character Mnemonics", APL Quote-Quad (Fall, 1974), vol. 5, No. 2, pp. 3-9.

[9] A. D. Falkoff and K. E. Iverson, APLSV User's Manual, IBM Publication SH20-1460 (1973).

[10] R. A. Kelley, "APLGOL, an Experimental Structured Programming Language", IBM J. Res. Develop. (January, 1973), vol. 17, pp. 69-73.

[11] M. A. Jenkins, A Control Structure Extension to APL, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Technical Report No. 21, (September, 1973), 13 pp.

[12] James R. Bell, "Threaded Code", Communications of the ACM (June, 1973), vol. 16, No. 6, pp. 370-372.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.